

Developer Studio 2006 Reference

**Delphi Language Guide
C++ Language Guide
Together Reference**

Borland®
Excellence Endures

Borland Software Corporation 100 Enterprise Way Scotts Valley, California 95066-3249 www.borland.com

Refer to the file `deploy.html` for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright 1997 2005 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

October 2005

PDF

Delphi Language Guide

Delphi Language Guide

Delphi Overview	21
Language Overview	23
Programs and Units	29
Programs and Units	31
Using Namespaces with Delphi	39
Fundamental Syntactic Elements	43
Fundamental Syntactic Elements	45
Declarations and Statements	51
Expressions	69
Data Types, Variables, and Constants	81
Data Types, Variables, and Constants	83
Simple Types	85
String Types	93
Structured Types	99
Pointers and Pointer Types	111
Procedural Types	115
Variant Types	119
Type Compatibility and Identity	123
Declaring Types	127
Variables	129
Declared Constants	131
Procedures and Functions	137
Procedures and Functions	139
Parameters	149
Calling Procedures and Functions	159
Classes and Objects	163
Classes and Objects	165
Fields	171
Methods	173
Properties	185
Events	193
Class References	197
Exceptions	201
Nested Type Declarations	209
Operator Overloading	211
Class Helpers	215
Standard Routines and I/O	217
Standard Routines and I/O	219
Libraries and Packages	229
Libraries and Packages	231
Writing Dynamically Loaded Libraries	233
Packages	239
Object Interfaces	243
Object Interfaces	245
Implementing Interfaces	249
Interface References	253
Automation Objects (Win32 Only)	255
Memory Management	259
Memory Management on the Win32 Platform	261
Internal Data Formats	263
Memory Management Issues on the .NET Platform	273

Program Control	277
Program Control	279
Inline Assembly Code (Win32 Only)	283
Using Inline Assembly Code (Win32 Only)	285
Understanding Assembler Syntax (Win32 Only)	287
Assembly Expressions (Win32 Only)	293
Assembly Procedures and Functions (Win32 Only)	303
Using .NET Custom Attributes	305

C++ Language Guide

C++ Language Guide

Lexical Elements	311
Lexical Elements	311
Whitespace Overview	313
Whitespace	313
Comments	313
Tokens Overview	317
Tokens	317
Keywords Overview	319
Keywords	319
C++-Specific Keywords	319
Table Of Borland C++ Register Pseudovariables	319
Keyword Extensions	320
Identifiers Overview	321
Identifiers	321
Constants Overview	323
Constants	323
Integer Constants	324
__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types	325
Integer Constant Without L Or U	325
Floating Point Constants	326
Character Constants Overview	329
Character Constants	329
The Three Char Types	330
Escape Sequences	330
Wide-character And Multi-character Constants	331
String Constants	333
String Constants	333
Enumeration Constants	335
Enumeration Constants	335
Constants and Internal Representation	337
Constants And Internal Representation	337
Internal Representation of Numerical Types	339
Internal Representation Of Numerical Types	339
Constant Expressions	341
Constant Expressions	341
Punctuators Overview	343
Punctuators	343
Language Structure	347
Language Structure	347
Declarations	349
Declarations	349
Objects	351
Objects	351
Storage Classes And Types	353
Storage Classes And Types	353
Scope	355
Scope	355
Visibility	357
Visibility	357
Duration	359
Duration	359

Static	359
Translation Units	361
Translation Units	361
Linkage	363
Linkage	363
Declaration Syntax	365
Declaration Syntax	365
Tentative Definitions	365
Possible Declarations	366
External Declarations And Definitions	368
Type Specifiers	369
Type Categories	371
Type Categories	371
Void	372
The Fundamental Types	373
The Fundamental Types	373
Initialization	377
Initialization	377
Declarations And Declarators	379
Declarations And Declarators	379
Use Of Storage Class Specifiers	381
Storage Class Specifiers	381
Variable Modifiers	383
Variable Modifiers	383
Const	383
Volatile	384
Mixed-Language Calling Conventions	387
Cdecl, __cdecl, __cdecl	387
Pascal, __pascal, __pascal	387
_stdcall, __stdcall	387
_fastcall, __fastcall	388
Multithread Variables	389
__thread, Multithread Variables	389
Function Modifiers	391
Function Modifiers	391
Pointers	393
Pointers	393
Pointers To Objects	393
Pointers To Functions	393
Pointer Declarations	394
Pointer Constants	394
Pointer Arithmetic	395
Pointer Conversions	395
C++ Reference Declarations	396
Arrays	396
Functions	399
Functions	399
Declarations And Definitions	399
Declarations And Prototypes	399
Definitions	401
Formal Parameter Declarations	401
Function Calls And Argument Conversions	402
Structures	403
Structures	403
Untagged Structures And Typedefs	403

Structure Member Declarations	404
Structures And Functions	404
Structure Member Access	404
Structure Name Spaces	406
Incomplete Declarations	406
Bit Fields	406
Unions	411
Unions	411
Anonymous Unions	411
Union Declarations	412
Enumerations	413
Enumerations	413
Assignment To Enum Types	414
Expressions	417
Expressions	417
Precedence Of Operators	419
Expressions And C++	421
Evaluation Order	421
Errors And Overflows	422
Operators Summary	423
Operators Summary	423
Primary Expression Operators	425
Primary Expression Operators	425
Postfix Expression Operators	427
Array Subscript Operator	427
Function Call Operator	427
. (direct Member Selector)	427
-> (indirect Member Selector)	428
Increment/decrement Operators	428
Unary Operators	431
Unary Operators	431
Reference/deference Operators	431
Plus And Minus Operators	432
Arithmetic Operators	433
Sizeof	434
Binary Operators	437
Binary Operators	437
Multiplicative Operators	438
Bitwise Operators	439
Relational Operators	439
Equality Operators	440
Logical Operators	440
Conditional Operators	441
Assignment Operators	441
Comma Operator	442
C++ Specific Operators	442
Statements	445
Statements	445
Blocks	446
Labeled Statements	446
Expression Statements	447
Selection Statements	447
Iteration Statements	447
Jump Statements	447
C++ Specifics	449

C++ Specifics	449
C++ namespaces	451
Defining A namespace	451
Declaring A namespace	451
namespace Alias	452
Extending A namespace	452
Anonymous namespaces	452
Accessing Elements Of A namespace	452
Using Directive	453
Explicit Access Qualification	453
New-style Typecasting Overview	455
New-style Typecasting	455
const_cast (typecast Operator)	455
dynamic_cast (typecast Operator)	455
reinterpret_cast (typecast Operator)	456
static_cast (typecast Operator)	456
Run-time Type Identification (RTTI)	459
Runtime Type Identification (RTTI) Overview	459
The Typeid Operator	461
__rtti, -RT Option	461
Runtime Type Identification And Destructors	462
Referencing	463
Referencing	463
Simple References	463
Reference Arguments	463
The Scope Resolution Operator	467
Scope Resolution Operator ::	467
The new And delete Operators	469
new	469
delete	470
Operator new Placement Syntax	470
Handling Errors For The New Operator	470
The Operator new With Arrays	470
The delete Operator With Arrays	471
Operator new	471
Overloading The Operator new	471
Overloading The Operator delete	472
Classes	473
C++ Classes	473
CLX and VCL Class Declarations	473
Class Names	474
Class Types	474
Class Name Scope	474
Class Objects	475
Class Member List	475
Member Functions	476
The Keyword This	477
Static Members	477
Inline Functions	479
Inline Functions	479
Member Scope	481
Member Scope	481
Nested Types	481
Member Access Control	482
Base And Derived Class Access	483

Virtual Base Classes	487
Virtual Base Classes	487
Friends Of Classes	489
Friends Of Classes	489
Constructors And Destructors	491
Introduction To Constructors And Destructors	491
Constructors	493
Constructors	493
Constructor Defaults	493
The Copy Constructor	494
Overloading Constructors	494
Order Of Calling Constructors	495
Class Initialization	496
Destructors	499
Destructors	499
Invoking Destructors	499
Atexit, #pragma Exit, And Destructors	499
Exit And Destructors	500
Abort And Destructors	500
Virtual Destructors	500
Operator Overloading Overview	503
Overloading Operators	503
How To Construct A Class Of Complex Vectors	505
Example Of Overloading Operators	505
Overloading Operator Functions Overview	509
Overloading Operator Functions	509
Overloaded Operators And Inheritance	509
Overloading Unary Operators	509
Overloading Binary Operators	510
Overloading The Assignment Operator =	510
Overloading The Function Call Operator ()	510
Overloading The Subscript Operator []	511
Overloading The Class Member Access Operators ->	511
Polymorphic Classes	513
Polymorphic Classes	513
Virtual Functions	515
Virtual Functions	515
Dynamic Functions	517
Abstract Classes	521
Abstract Classes	521
C++ Scope	523
C++ Scope	523
Class Scope	523
Hiding	523
C++ Scoping Rules Summary	523
Templates	525
Using Templates	525
template	525
Template Body Parsing	526
Function Templates Overview	527
Function Templates	527
Overriding A Template Function	527
Implicit And Explicit Template Functions	528
Class Templates Overview	529
Class Templates	529

Template Arguments	529
Using Angle Brackets In Templates	529
Using Type-safe Generic Lists In Templates	530
Eliminating Pointers In Templates	530
Compiler Template Switches	533
Template Compiler Switches	533
Template Generation Semantics	533
Exporting And Importing Templates	535
Exporting And Importing Templates	535
The Preprocessor	537
Preprocessor Directives	539
Preprocessor Directives	539
# (null Directive)	539
Defining And Undefining Macros	541
#define	541
#undef	541
Using The -D And -U Command-line Options	542
Keywords And Protected Words In Macros	542
Macros With Parameters Overview	543
Macros With Parameters	543
Nesting Parentheses And Commas	543
Token Pasting With ##	544
Converting To Strings With #	544
Using The Backslash (\) For Line Continuation	544
Side Effects And Other Dangers	544
File Inclusion With #include	547
#include	547
Header File Search With <header_name>	547
Header File Search With "header_name"	547
Conditional Compilation Overview	549
Conditional Compilation	549
Defined	549
#if, #elif, #else, And #endif	549
#ifdef And #ifndef	550
The #line Control Directive	553
#line	553
The #error Control Directive	555
#error	555
Pragma Directives Overview	557
#pragma Summary	557
#pragma Anon_struct	558
#pragma Argsused	559
#pragma Codeseg	559
#pragma Comment	559
#pragma exit and #pragma startup	559
#pragma Hdrfile	560
#pragma Hdrstop	560
#pragma Inline	560
#pragma Intrinsic	561
#pragma Link	561
#pragma Message	561
#pragma Pack	562
#pragma Package	563
#pragma Obsolete	564
#pragma Option	564

#pragma Resource	567
#pragma Warn	567
Predefined Macros Overview	569
Predefined Macros	569
Keywords, By Category	571
C++ Specific Keywords	573
Asm, _asm, __asm	573
Bool, False, True	574
Catch	574
Class	574
const_cast (typecast Operator)	455
delete	470
dynamic_cast (typecast Operator)	455
Explicit	576
Friend	577
Inline	577
Mutable	578
namespace	578
new	469
Operator	579
Private	579
Protected	580
Public	580
reinterpret_cast (typecast Operator)	456
_rtti, -RT Option	461
static_cast (typecast Operator)	456
template	525
This	583
Throw	584
Try	584
Typeid	584
Typename	585
using (declaration)	586
Virtual	586
Wchar_t	586
C++ Builder Keyword Extensions	587
Asm, _asm, __asm	573
__automated	588
Cdecl, _cdecl, __cdecl	387
__classid	589
__closure	589
__declspec	589
__declspec(dllexport)	590
__declspec(dllimport)	590
__declspec(naked)	591
__declspec(noreturn)	591
__declspec(nothrow)	592
__declspec(novtable)	592
__declspec(property)	593
__declspec(selectany)	593
__declspec(thread)	594
__declspec(uuid("ComObjectGUID"))	594
__except	595
__export, __export	595
__fastcall, __fastcall	388

__finally	596
__import, __import	598
__inline	599
__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types	325
__msfastcall	599
__msreturn	600
__thread, Multithread Variables	389
Pascal, __pascal, __pascal	387
__property	600
__published	601
__rtti, -RT Option	461
__stdcall, __stdcall	387
__try	602
Modifiers	605
Cdecl, __cdecl, __cdecl	387
Const	383
__declspec	589
__declspec(dllexport)	590
__declspec(dllimport)	590
__declspec(naked)	591
__declspec(noreturn)	591
__declspec(nothrow)	592
__declspec(novtable)	592
__declspec(property)	593
__declspec(selectany)	593
__declspec(thread)	594
__declspec(uuid("ComObjectGUID"))	594
__dispid	612
__export, __export	595
__fastcall, __fastcall	388
__import, __import	598
__msfastcall	599
__msreturn	600
Pascal, __pascal, __pascal	387
__rtti, -RT Option	461
__stdcall, __stdcall	387
Volatile	384
Operators	619
__classid	589
delete	470
If, Else	620
new	469
Operator	579
Sizeof	434
Typeid	584
Special Types	625
Void	372
Statements	627
Break	627
Case	627
Catch	574
Continue	628
Default	628
Do	629
__except	595

__finally	596
For	631
Goto	632
Return	632
Switch	632
Throw	584
__try	602
Try	584
While	634
Storage Class Specifiers	635
Auto	635
__declspec	589
__declspec(dllexport)	590
__declspec(dllimport)	590
__declspec(naked)	591
__declspec(noreturn)	591
__declspec(nothrow)	592
__declspec(novtable)	592
__declspec(property)	593
__declspec(selectany)	593
__declspec(thread)	594
__declspec(uuid("ComObjectGUID"))	594
Extern	641
Mutable	578
Register	641
Typedef	642
Type Specifiers	643
Char	643
Class	574
Double	644
Enum	644
Float	645
Int	645
Long	645
Short	646
Signed	646
Struct	646
Union	647
Unsigned	647
Wchar_t	586

Together Reference

Together Reference

Together Glossary	651
GUI Components for Modeling	653
Menus	655
Tool Palette	657
Object Inspector	659
Diagram View	661
Model View	663
Pattern GUI Components	665
Pattern Organizer	667
Pattern Registry	669
Quality Assurance GUI Components	671
Audit Results Pane	673
Metric Results Pane	675
Together Wizards	677
New Together Project Wizards	679
UML 1.5 Together Design Project Wizard	681
UML 2.0 Together Design Project Wizard	683
Convert MDL Wizard	685
Supported Delphi Project Wizards	687
Supported C# Project Wizards	689
Pattern Wizard	691
Create Pattern Wizard	693
Together Keyboard Shortcuts	695
Together Configuration Options	697
Configuration Levels	699
Option Value Editors	701
Together Option Categories	703
Together General Options	705
Together Diagram Appearance Options	707
Together Diagram Layout Options	711
Together Diagram Print Options	715
Together Diagram View Filters Options	719
Together Generate Documentation Options	721
Together Model View Options	723
Together Source Code Options	725
UML 1.5 Reference	727
UML 1.5 Class Diagrams	729
UML 1.5 Class Diagram Definition	731
Class Diagram Types	733
UML 1.5 Class Diagram Elements	735
Class Diagram Relationships	737
LiveSource Rules	739
Association Class and N-ary Association	741
Inner Classifiers	743
Members	745
UML 1.5 Use Case Diagrams	747
UML 1.5 Use Case Diagram Definition	749
UML 1.5 Use Case Diagram Elements	751
Extension Point	753
UML 1.5 Interaction Diagrams	755
UML 1.5 Sequence Diagram Definition	757

UML 1.5 Collaboration Diagram Definition	759
UML 1.5 Interaction Diagram Elements	761
Conditional Block	763
UML 1.5 Message	765
Activation Bar	767
Nested Message	769
UML 1.5 Statechart Diagrams	771
UML 1.5 Statechart Diagram Definition	773
UML 1.5 Statechart Diagram Elements	775
State	777
Transition	779
Deferred Event	781
UML 1.5 Activity Diagrams	783
UML 1.5 Activity Diagram Elements	785
UML 1.5 Activity Diagram Definition	787
State	777
Transition	779
Deferred Event	781
UML 1.5 Component Diagrams	795
UML 1.5 Component Diagram Definition	797
UML 1.5 Component Diagram Elements	799
UML 1.5 Deployment Diagrams	801
UML 1.5 Deployment Diagram Definition	803
UML 1.5 Deployment Diagram Elements	805
UML 2.0 Reference	807
UML 2.0 Class Diagrams	809
UML 2.0 Class Diagram Definition	811
Class Diagram Types	733
UML 2.0 Class Diagram Elements	815
Class Diagram Relationships	737
Association Class and N-ary Association	741
Inner Classifiers	743
Members	745
UML 2.0 Use Case Diagrams	825
UML 2.0 Use Case Diagram Definition	827
UML 2.0 Use Case Diagram Elements	829
Extension Point	753
UML 2.0 Interaction Diagrams	833
UML 2.0 Sequence Diagram Definition	835
UML 2.0 Communication Diagram Definition	837
UML 2.0 Sequence Diagram Elements	839
UML 2.0 Communication Diagram Elements	841
Interaction	843
UML 2.0 Message	845
Execution Specification and Invocation Specification	847
Operator and Operand for a Combined Fragment	849
UML 2.0 State Machine Diagrams	851
UML 2.0 State Machine Diagram Definition	853
UML 2.0 State Machine Diagram Elements	855
State	777
Transition	779
Deferred Event	781
History Element (State Machine Diagrams)	863
UML 2.0 Activity Diagrams	865
UML 2.0 Activity Diagram Definition	867

UML 2.0 Activity Diagram Elements	869
Pin	871
UML 2.0 Component Diagrams	873
UML 2.0 Component Diagram Definition	875
UML 2.0 Component Diagram Elements	877
UML 2.0 Deployment Diagrams	879
UML 2.0 Deployment Diagram Definition	881
UML 2.0 Deployment Diagram Elements	883
UML 2.0 Composite Structure Diagrams	885
UML 2.0 Composite Structure Diagram Definition	887
UML 2.0 Composite Structure Diagram Elements	889
Delegation Connector	891
Together Refactoring Operations	893
Project Types and Formats with Support for Modeling	895

Delphi Language Guide

The Delphi Language guide describes the Delphi language as it is used in Borland development tools. This book describes the Delphi language on both the Win32, and .NET development platforms. Specific differences in the language between the two platforms are marked as appropriate.

In This Section

[Language Overview](#)

Describes high-level Delphi language concepts such as program organization, source file naming conventions. Also covers basic command-line compiler usage, with short examples.

[Programs and Units](#)

Describes the structure of Delphi programs in detail, covering project files, units, and namespaces.

[Fundamental Syntactic Elements](#)

Describes the syntax for identifiers, character strings, numbers, and labels.

[Data Types, Variables, and Constants](#)

A conceptual overview of data types in the Delphi language.

[Procedures and Functions](#)

Describes the basics of procedure and function declarations in Delphi.

[Classes and Objects](#)

Presents a conceptual overview of classes and class types in the Delphi language.

[Standard Routines and I/O](#)

Describes text and file I/O and standard library routines.

[Libraries and Packages](#)

Describes the use of libraries and packages.

[Object Interfaces](#)

Describes the declaration and implementation of interfaces in Delphi.

[Memory Management on the Win32 Platform](#)

Describes how programs use memory on the Win32 platform.

[Program Control](#)

Describes parameters, functions, method calls, and exit procedures.

[Using Inline Assembly Code \(Win32 Only\)](#)

Describes the inline assembler, which is available in the Win32 Delphi compiler only.

Delphi Overview

This chapter provides a brief introduction to Delphi programs, and program organization.

In This Section

[Language Overview](#)

Describes high-level Delphi language concepts such as program organization, source file naming conventions. Also covers basic command-line compiler usage, with short examples.

Language Overview

Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming. Delphi has special features that support Borland's component framework and RAD environment. For the most part, descriptions and examples in this language guide assume that you are using Borland development tools.

Most developers using Borland software development tools write and compile their code in the integrated development environment (IDE). Borland development tools handle many details of setting up projects and source files, such as maintenance of dependency information among units. The product also places constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, Borland development tools enforce certain file- and program-naming conventions that you can avoid if you write your programs outside of the IDE and compile them from the command prompt.

This language guide generally assumes that you are working in the IDE and that you are building applications that use the Borland Visual Component Library (VCL). Occasionally, however, Delphi-specific rules are distinguished from rules that apply to all Object Pascal programming. This text covers both the Win32 Delphi language compiler, and the Delphi for .NET language compiler. Platform-specific language differences and features are noted where necessary.

This section covers the following topics:

- **Program Organization.** Covers the basic language features that allow you to partition your application into units and namespaces.
- **Example Programs.** Small examples of both console and GUI applications are shown, with basic instructions on running the compiler from the command-line.

Program Organization

Delphi programs are usually divided into source-code modules called units. Most programs begin with a program heading, which specifies a name for the program. The program heading is followed by an optional uses clause, then a block of declarations and statements. The uses clause lists units that are linked into the program; these units, which can be shared by different programs, often have uses clauses of their own.

The uses clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, most Delphi language programs do not require makefiles, header files, or preprocessor "include" directives.

Delphi Source Files

The compiler expects to find Delphi source code in files of three kinds:

- Unit source files (which end with the .pas extension)
- Project files (which end with the .dpr extension)
- Package source files (which end with the .dpk extension)

Unit source files typically contain most of the code in an application. Each application has a single project file and several unit files; the project file, which corresponds to the program file in traditional Pascal, organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. If you use the IDE to build your application, it will produce a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called packages.

Other Files Used to Build Applications

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically by the IDE, and include

- VCL form files (which have a .dfm extension on Win32, and .nfm on .NET)
- Resource files (which end with .res)
- Project options files (which end with .dof)

A VCL form file contains the description of the properties of the form and the components it owns. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text (a format very suitable for version control) or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to VCL form files, each project uses a resource (.res) file to hold the application's icon and other resources such as strings. By default, this file has the same name as the project (.dpr) file.

A project options (.dof) file contains compiler and linker settings, search path information, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

Compiler-Generated Files

The first time you build an application or a package, the compiler produces a compiled unit file (.dcu on Win32, .dcuil on .NET) for each new unit used in your project; all the .dcu/.dcuil files in your project are then linked to create a single executable or shared package. The first time you build a package, the compiler produces a file for each new unit contained in the package, and then creates both a .dcp and a package file. If you use the **GD** switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you build a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, their .dcu/.dpu files cannot be found, you explicitly tell the compiler to reprocess them, or the interface of the unit depends on another unit which has been changed. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file and that unit has no dependencies on other units that have changed.

Example Programs

The examples that follow illustrate basic features of Delphi programming. The examples show simple applications that would not normally be compiled from the IDE; you can compile them from the command line.

A Simple Console Application

The program below is a simple console application that you can compile and run from the command prompt.

```
program greeting;  
  
{$APPTYPE CONSOLE}  
  
var MyMessage: string;
```



```
begin
    MyMessage := 'Hello world!';
    Writeln(MyMessage);
end.
```

The first line declares a program called Greeting. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called `MyMessage`, which holds a string. (Delphi has genuine string data types.) The program then assigns the string "Hello world!" to the variable `MyMessage`, and sends the contents of `MyMessage` to the standard output using the `Writeln` procedure. (`Writeln` is defined implicitly in the `System` unit, which the compiler automatically includes in every application.)

You can type this program into a file called `greeting.pas` or `greeting.dpr` and compile it by entering

`dcc32 greeting`

to produce a Win32 executable, or

`dccil greeting`

to produce a managed .NET executable. In either case, the resulting executable prints the message Hello world!

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Borland development tools. First, it is a console application. Borland development tools are most often used to write applications with graphical interfaces; hence, you would not ordinarily call `Writeln`. Moreover, the entire example program (save for `Writeln`) is in a single file. In a typical GUI application, the program heading the first line of the example would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to routines defined in unit files.

A More Complicated Example

The next example shows a program that is divided into two files: a project file and a unit file. The project file, which you can save as `greeting.dpr`, looks like this:

```
program greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
    PrintMessage('Hello World!');
end.
```

The first line declares a program called `greeting`, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that the program `greeting` depends on a unit called `Unit1`. Finally, the program calls the `PrintMessage` procedure, passing to it the string Hello World! The `PrintMessage` procedure is defined in `Unit1`. Here is the source code for `Unit1`, which must be saved in a file called `Unit1.pas`:

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation
```

```

procedure PrintMessage(msg: string);
begin
    Writeln(msg);
end;

end.

```

`Unit1` defines a procedure called `PrintMessage` that takes a single string as an argument and sends the string to the standard output. (In Delphi, routines that do not return a value are called procedures. Routines that return a value are called functions.) Notice that `PrintMessage` is declared twice in `Unit1`. The first declaration, under the reserved word `interface`, makes `PrintMessage` available to other modules (such as `greeting`) that use `Unit1`. The second declaration, under the reserved word `implementation`, actually defines `PrintMessage`.

You can now compile `Greeting` from the command line by entering

`dcc32 greeting`

to produce a Win32 executable, or

`dccil greeting`

to produce a managed .NET executable.

There is no need to include `Unit1` as a command-line argument. When the compiler processes `greeting.dpr`, it automatically looks for unit files that the `greeting` program depends on. The resulting executable does the same thing as our first example: it prints the message `Hello world!`

A VCL Application

Our next example is an application built using the Visual Component Library (VCL) components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of the Delphi Language. In addition to multiple units, the program uses classes and objects

The program includes a project file and two new unit files. First, the project file:

```

program greeting;

    uses Forms, Unit1, Unit2;
    {$R *.res} // This directive links the project's resource file.

begin
    // Calls to global Application instance
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.CreateForm(TForm2, Form2);
    Application.Run;
end.

```

Once again, our program is called `greeting`. It uses three units: `Forms`, which is part of VCL; `Unit1`, which is associated with the application's main form (`Form1`); and `Unit2`, which is associated with another form (`Form2`).

The program makes a series of calls to an object named `Application`, which is an instance of the `TApplication` class defined in the `Forms` unit. (Every project has an automatically generated `Application` object.) Two of these calls invoke a `TApplication` method named `CreateForm`. The first call to `CreateForm` creates `Form1`, an instance of the `TForm1` class defined in `Unit1`. The second call to `CreateForm` creates `Form2`, an instance of the `TForm2` class defined in `Unit2`.

Unit1 looks like this:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type

TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
end;

var
    Form1: TForm1;

implementation

uses Unit2;

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
    Form2.ShowModal;
end;

end.
```

Unit1 creates a class named `TForm1` (derived from `TForm`) and an instance of this class, `Form1`. `TForm1` includes a button `Button1`, an instance of `TButton` and a procedure named `Button1Click` that is called when the user presses `Button1`. `Button1Click` hides `Form1` and it displays `Form2` (the call to `Form2.ShowModal`).

Note: In the previous example, `Form2.ShowModal` relies on the use of auto-created forms. While this is fine for example code, using auto-created forms is actively discouraged.

`Form2` is defined in `Unit2`:

```
unit Unit2;

interface

uses SysUtils, Types, Classes, Graphics, Controls, Forms, Dialogs;

type

TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
end;

var
    Form2: TForm2;

implementation
```

```
uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
    Form2.Close;
end;

end.
```

`Unit2` creates a class named `TForm2` and an instance of this class, `Form2`. `TForm2` includes a button (`CancelButton`, an instance of `TButton`) and a label (`Label1`, an instance of `TLabel`). You can't see this from the source code, but `Label1` displays a caption that reads Hello world! The caption is defined in `Form2`'s form file, `Unit2.dfm`.

`TForm2` declares and defines a method `CancelButtonClick` which will be invoked at runtime whenever the user presses `CancelButton`. This procedure (along with `Unit1`'s `TForm1.Button1Click`) is called an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form files for `Form1` and `Form2`.

When the `greeting` program starts, `Form1` is displayed and `Form2` is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's main form.) When the user presses the button on `Form1`, `Form2`, displays the Hello world! greeting. When the user presses the `CancelButton` or the `Close` button on the title bar, `Form2` closes.

Programs and Units

This chapter provides a more detailed look at Delphi program organization.

In This Section

[Programs and Units](#)

Describes the structure of Delphi programs in detail, covering project files, units, and namespaces.

[Using Namespaces with Delphi](#)

Describes the use of .NET namespaces in Delphi applications.

Programs and Units

A Delphi program is constructed from source code modules called units. The units are tied together by a special source code module that contains either the program, library, or package header. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. Developer Studio 2006 introduces hierarchical namespaces, giving you even more flexibility in organizing your units. Namespaces and units allow you to

- Divide large programs into modules that can be edited separately.
- Create libraries that you can share among programs.
- Distribute libraries to other developers without making the source code available.

This topic covers the overall structure of a Delphi application: the program header, unit declaration syntax, and the uses clause. Specific differences between the Win32 and .NET platforms are noted in the text. The Delphi compiler does not support .NET namespaces on the Win32 platform. The Developer Studio 2006 compiler does support hierarchical .NET namespaces; this topic is covered in the following section, Using Namespaces with Delphi.

Program Structure and Syntax

A complete, executable Delphi application consists of multiple unit modules, all tied together by a single source code module called a project file. In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use the file extension .dpr to designate the main program source module, while most other source code resides in unit files having the traditional .pas extension. To build a project, the compiler needs the project source file, and either a source file or a compiled unit file for each unit.

Note: Strictly speaking, you need not explicitly use any units in a project, but all programs automatically use the `System` unit and the `SysInit` unit.

The source code file for an executable Delphi application contains

- a program heading,
- a uses clause (optional), and
- a block of declarations and executable statements.

Additionally, a Developer Studio 2006 program may contain a namespaces clause, to specify additional namespaces in which to search for generic units. This topic is covered in more detail in the section Using .NET Namespaces with Delphi.

The compiler, and hence the IDE, expect to find these three elements in a single project (.dpr) file.

The Program Heading

The program heading specifies a name for the executable program. It consists of the reserved word `program`, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the project source file name.

The following example shows the project source file for a program called Editor. Since the program is called Editor, this project file is called Editor.dpr.

```
program Editor;

uses Forms, REAbout, // An "About" box
    REMain;           // Main form
```

```
{ $R *.res }

begin
  Application.Title := 'Text Editor';
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

The first line contains the program heading. The `uses` clause in this example specifies a dependency on three additional units: `Forms`, `REAbout`, and `REMain`. The `$R` compiler directive links the project's resource file into the program. Finally, the block of statements between the `begin` and `end` keywords are executed when the program runs. The project file, like all Delphi source files, ends with a period (not a semicolon).

Delphi project files are usually short, since most of a program's logic resides in its unit files. A Delphi project file typically contains only enough code to launch the application's main window, and start the event processing loop. Project files are generated and maintained automatically by the IDE, and it is seldom necessary to edit them manually.

In standard Pascal, a program heading can include parameters after the program name:

```
program Calc(input, output);
```

Borland's Delphi ignores these parameters.

In Developer Studio 2006, a the program heading introduces its own namespace, which is called the project default namespace. This is also true for the library and package headers, when these types of projects are compiled for the .NET platform.

The Program Uses Clause

The `uses` clause lists those units that are incorporated into the program. These units may in turn have `uses` clauses of their own. For more information on the `uses` clause within a unit source file, see *Unit References and the Uses Clause*, below.

The `uses` clause consists of the keyword `uses`, followed by a comma delimited list of units the project file directly depends on.

The Block

The block contains a simple or structured statement that is executed when the program runs. In most program files, the block consists of a compound statement bracketed between the reserved words `begin` and `end`, whose component statements are simply method calls to the project's `Application` object. Most projects have a global `Application` variable that holds an instance of `TApplication`, `TWebApplication`, or `TServiceApplication`. The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

Unit Structure and Syntax

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own source (.pas) file.

A unit file begins with a unit heading, which is followed by the `interface` keyword. Following the `interface` keyword, the `uses` clause specifies a list of unit dependencies. Next comes the implementation section, followed by the optional initialization, and finalization sections. A skeleton unit source file looks like this:

```
unit Unit1;
```



```

interface

uses // List of unit dependencies goes here...

implementation

uses // List of unit dependencies goes here...

// Implementation of class methods, procedures, and functions goes here...

initialization

// Unit initialization code goes here...

finalization

// Unit finalization code goes here...

end.

```

The unit must conclude with the reserved word `end` followed by a period.

The Unit Heading

The unit heading specifies the unit's name. It consists of the reserved word `unit`, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the unit file name. Thus, the unit heading

```
unit MainForm;
```

would occur in a source file called `MainForm.pas`, and the file containing the compiled unit would be `MainForm.dcu` or `MainForm.dpu`.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

The Interface Section

The interface section of a unit begins with the reserved word `interface` and continues until the beginning of the implementation section. The interface section declares constants, types, variables, procedures, and functions that are available to clients. That is, to other units or programs that wish to use elements from this unit. These entities are called *public* because code in other units can access them as if they were declared in the unit itself.

The interface declaration of a procedure or function includes only the routine's signature. That is, the routine's name, parameters, and return type (for functions). The block containing executable code for the procedure or function follows in the implementation section. Thus procedure and function declarations in the interface section work like forward declarations.

The interface declaration for a class must include declarations for all class members: fields, properties, procedures, and functions.

The interface section can include its own `uses` clause, which must appear immediately after the keyword `interface`.

The Implementation Section

The implementation section of a unit begins with the reserved word `implementation` and continues until the beginning of the initialization section or, if there is no initialization section, until the end of the unit. The implementation section defines procedures and functions that are declared in the interface section. Within the implementation section, these

procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the implementation section; but if you include a parameter list, it must match the declaration in the interface section exactly.

In addition to definitions of public procedures and functions, the implementation section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit. That is, unlike the interface section, entities declared in the implementation section are inaccessible to other units.

The implementation section can include its own uses clause, which must appear immediately after the keyword `implementation`. The identifiers declared within units specified in the implementation section are only available for use within the implementation section itself. You cannot refer to such identifiers in the interface section.

The Initialization Section

The initialization section is optional. It begins with the reserved word `initialization` and continues until the beginning of the finalization section or, if there is no finalization section, until the end of the unit. The initialization section contains statements that are executed, in the order in which they appear, on program start-up. So, for example, if you have defined data structures that need to be initialized, you can do this in the initialization section.

For units in the interface uses list, the initialization sections of the units used by a client are executed in the order in which the units appear in the client's uses clause.

The Finalization Section

The finalization section is optional and can appear only in units that have an initialization section. The finalization section begins with the reserved word `finalization` and continues until the end of the unit. It contains statements that are executed when the main program terminates (unless the *Halt* procedure is used to terminate the program). Use the finalization section to free resources that are allocated in the initialization section.

Finalization sections are executed in the opposite order from initialization sections. For example, if your application initializes units A, B, and C, in that order, it will finalize them in the order C, B, and A.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the initialization code might not execute completely.

Note: The initialization and finalization sections behave differently when code is compiled for the managed .NET environment. See the chapter on Memory Management for more information.

Unit References and the Uses Clause

A uses clause lists units used by the program, library, or unit in which the clause appears. A uses clause can occur in

- the project file for a program, or library
- the interface section of a unit
- the implementation section of a unit

Most project files contain a uses clause, as do the interface sections of most units. The implementation section of a unit can contain its own uses clause as well.

The `System` unit and the `SysInit` unit are used automatically by every application and cannot be listed explicitly in the uses clause. (`System` implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as `SysUtils`, must be explicitly included in the uses clause. In most cases, all necessary units are placed in the uses clause by the IDE, as you add and remove units from your project.

In unit declarations and uses clauses, unit names must match the file names in case. In other contexts (such as qualified identifiers), unit names are case insensitive. To avoid problems with unit references, refer to the unit source file explicitly:

```
uses MyUnit in "myunit.pas";
```

If such an explicit reference appears in the project file, other source files can refer to the unit with a simple uses clause that does not need to match case:

```
uses Myunit;
```

The Syntax of a Uses Clause

A uses clause consists of the reserved word `uses`, followed by one or more comma delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;

uses
    Forms,
    Main;

uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
```

In the uses clause of a program or library, any unit name may be followed by the reserved word `in` and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

```
uses
    Windows, Messages, SysUtils,
    Strings in 'C:\Classes\Strings.pas', Classes;
```

Use the keyword `in` after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no reason to do this. Using `in` is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the `in ...` construction to determine which units are part of a project. Only units that appear in a project (.dpr) file's uses clause followed by `in` and a file name are considered to be part of the project; other units in the uses clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the **Project Manager**.

In the uses clause of a unit, you cannot use `in` to tell the compiler where to find a source file. Every unit must be in the compiler's search path. Moreover, unit names must match the names of their source files.

Multiple and Indirect Unit References

The order in which units appear in the uses clause determines the order of their initialization and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the same name, the compiler uses the one from the unit listed last in the uses clause. (To access the identifier from the other unit, you would have to add a qualifier: `UnitName.Identifier`.)

A uses clause need include only units used directly by the program or unit in which the clause appears. That is, if unit A references constants, types, variables, procedures, or functions that are declared in unit B, then A must use B explicitly. If B in turn references identifiers from unit C, then A is indirectly dependent on C; in this case, C needn't be included in a uses clause in A, but the compiler must still be able to find both B and C in order to process A.

The following example illustrates indirect dependency.

```
program Prog;
uses Unit2;
const a = b;
// ...

unit Unit2;
interface
uses Unit1;
const b = c;
// ...

unit Unit1;
interface
const c = 1;
// ...
```

In this example, `Prog` depends directly on `Unit2`, which depends directly on `Unit1`. Hence `Prog` is indirectly dependent on `Unit1`. Because `Unit1` does not appear in `Prog`'s **uses** clause, identifiers declared in `Unit1` are not available to `Prog`.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their `.dcu` (Win32) or `.dcuil` (.NET) files, not their source (`.pas`) files.

When a change is made in the interface section of a unit, other units that depend on the change must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

Circular Unit References

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the uses clause of one interface section to the uses clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the uses clause of at least one implementation section.

In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface uses clauses. So the following example leads to a compilation error:

```
unit Unit1;
interface
uses Unit2;
```

```
// ...  
  
unit Unit2;  
interface  
uses Unit1;  
// ...
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;  
interface  
uses Unit2;  
// ...  
  
unit Unit2;  
interface  
//...  
  
implementation  
uses Unit1;  
// ...
```

To reduce the chance of circular references, it's a good idea to list units in the implementation uses clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface uses clause.

Using Namespaces with Delphi

In Delphi, a unit is the basic container for types. Microsoft's Common Language Runtime (CLR) introduces another layer of organization called a namespace. In the .NET Framework, a namespace is a conceptual container of types. In Delphi, a namespace is a container of Delphi units. The addition of namespaces gives Delphi the ability to access and extend classes in the .NET Framework.

Unlike traditional Delphi units, namespaces can be nested to form a containment hierarchy. Nested namespaces provide a way to organize identifiers and types, and are used to disambiguate types with the same name. Since they are a container for Delphi units, namespaces may also be used to differentiate between units of the same name, that reside in different packages.

For example, the class `MyClass` in `MyNameSpace`, is different from the class `MyClass` in `YourNameSpace`. At runtime, the CLR always refers to classes and types by their fully qualified names: the assembly name, followed by the namespace that contains the type. The CLR itself has no concept or implementation of the namespace hierarchy; it is purely a notational convenience of the programming language.

The following topics are covered:

- Project default namespaces, and namespace declaration.
- Namespace search scope.
- Using namespaces in Delphi units.

Declaring Namespaces

In Developer Studio 2006, a project file (program, library, or package) implicitly introduces its own namespace, called the *project default namespace*. A unit may be a member of the project default namespace, or it may explicitly declare itself to be a member of a different namespace. In either case, a unit declares its namespace membership in its unit header. For example, consider the following explicit namespace declaration:

```
unit MyCompany.MyWidgets.MyUnit;
```

First, notice that namespaces are separated by dots. Namespaces do not introduce new symbols for the identifiers between the dots; the dots are part of the unit name. The source file name for this example is `MyCompany.MyWidgets.MyUnit.pas`, and the compiled output file name is `MyCompany.MyWidgets.MyUnit.dcuil`.

Second, notice that the dots imply the conceptual nesting, or containment, of one namespace within another. The example above declares the unit `MyUnit` to be a member of the `MyWidgets` namespace, which itself is contained in the `MyCompany` namespace. Again, it should be noted that this containment is for documentation purposes only.

A project default namespace declares a namespace for all of the units in the project. Consider the following declarations:

```
Program MyCompany.Programs.MyProgram;  
Library MyCompany.Libs.MyLibrary;  
Package MyCompany.Packages.MyPackage;
```

These statements establish the project default namespace for the program, library, and package, respectively. The namespace is determined by removing the rightmost identifier (and dot) from the declaration.

A unit that omits an explicit namespace is called a *generic unit*. A generic unit automatically becomes a member of the project default namespace. Given the preceding program declaration, the following unit declaration would cause the compiler to treat `MyUnit` as a member of the `MyCompany.Programs` namespace.

```
unit MyUnit;
```

The project default namespace does not affect the name of the Delphi source file for a generic unit. In the preceding example, the Delphi source file name would be `MyUnit.pas`. The compiler does however prefix the `dcuil` file name with the project default namespace. The resulting `dcuil` file in the current example would be `MyCompany.Programs.MyUnit.dcuil`.

Namespace strings are not case-sensitive. The compiler considers two namespaces that differ only in case to be equivalent. However, the compiler does preserve the case of a namespace, and will use the preserved casing in output file names, error messages, and RTTI unit identifiers. RTTI for class and type names will include the full namespace specification.

Searching Namespaces

A unit must declare the other units on which it depends. As with the Win32 platform, the compiler must search these units for identifiers. For units in explicit namespaces the search scope is already known, but for generic units, the compiler must establish a namespace search scope.

Consider the following unit and uses declarations:

```
unit MyCompany.ProjectX.ProgramY.MyUnit1;  
uses MyCompany.Libs.Unit2, Unit3, Unit4;
```

These declarations establish `MyUnit1` as a member of the `MyCompany.ProjectX.ProgramY` namespace. `MyUnit1` depends on three other units: `MyCompany.Libs.Unit2`, and the generic units, `Unit3`, and `Unit4`. The compiler can resolve identifier names in `Unit2`, since the uses clause specified the fully qualified unit name. To resolve identifier names in `Unit3` and `Unit4`, the compiler must establish a namespace search order.

Namespace search order

Search locations can come from three possible sources: compiler options, the project default namespace, and the current unit's namespace.

The compiler resolves identifier names in the following order:

- 1 The current unit namespace (if any)
- 2 The project default namespace (if any)
- 3 Namespaces specified by compiler options.

A namespace search example

The following example project and unit files demonstrate the namespace search order:

```
// Project file declarations...  
program MyCompany.ProjectX.ProgramY;
```

```
// Unit source file declaration...  
unit MyCompany.ProjectX.ProgramY.MyUnit1;
```

Given this program example, the compiler would search namespaces in the following order:

- 1 `MyCompany.ProjectX.ProgramY`
- 2 `MyCompany.ProjectX`
- 3 Namespaces specified by compiler options.

Note that if the current unit is generic (i.e. it does not have an explicit namespace declaration in its unit statement), then resolution begins with the project default namespace.

Using Namespaces

Delphi's `uses` clause brings a module into the context of the current unit. The `uses` clause must either refer to a module by its fully qualified name (i.e. including the full namespace specification), or by its generic name, thereby relying on the namespace resolution mechanisms to locate the unit.

Fully qualified unit names

The following example demonstrates the `uses` clause with namespaces:

```
unit MyCompany.Libs.MyUnit1
uses MyCompany.Libs.Unit2,    // Fully qualified name.
     UnitX;                   // Generic name.
```

Once a module has been brought into context, source code can refer to identifiers within that module either by the unqualified name, or by the fully qualified name (if necessary, to disambiguate identifiers with the same name in different units). The following `writeln` statements are equivalent:

```
uses MyCompany.Libs.Unit2;

begin
    writeln(MyCompany.Libs.Unit2.SomeString);
    writeln(SomeString);
end.
```

A fully qualified identifier must include the full namespace specification. In the preceding example, it would be an error to refer to `SomeString` using only a portion of the namespace:

```
writeln(Unit2.SomeString);           // ERROR!
writeln(Libs.Unit2.SomeString);      // ERROR!
writeln(MyCompany.Libs.Unit2.SomeString); // Correct.
writeln(SomeString);                 // Correct.
```

It is also an error to refer to only a portion of a namespace in the `uses` clause. There is no mechanism to import all units and symbols in a namespace. The following code does not import all units and symbols in the `MyCompany` namespace:

```
uses MyCompany;    // ERROR!
```

This restriction also applies to the `with-do` statement. The following will produce a compiler error:

```
with MyCompany.Libs do    // ERROR!
```

Namespaces and .NET Metadata

The Delphi for .NET compiler does not emit the entire dotted unit name into the assembly. Instead, the only leftmost portion - everything up to the last dot in the name is emitted. For example:

```
unit MyCompany.MyClasses.MyUnit
```

The compiler will emit the namespace `MyCompany.MyClasses` into the assembly metadata. This makes it easier for other .NET languages to call into Delphi assemblies.

This difference in namespace metadata is visible only to external consumers of the assembly. The Delphi code within the assembly still treats the entire dotted name as the fully qualified name.

Multi-unit Namespaces

Multiple units can belong to the same namespace, if the unit declarations refer to the same namespace. For example, one can create two files, `unit1.pas` and `unit2.pas`, with the following unit declarations:

```
// in file 'unit1.pas'  
unit MyCompany.ProjectX.ProgramY.Unit1
```

```
// in file 'unit2.pas'  
unit MyCompany.ProjectX.ProgramY.Unit2
```

In this example, the namespace `MyCompany.ProjectX.ProgramY` logically contains all of the interface symbols from `unit1.pas` and `unit2.pas`.

Symbol names in a namespace must be unique, across all units in the namespace. In the example above, it is an error for `Unit1` and `Unit2` to both define a global interface symbol named `mySymbol`.

The individual units aggregated in a namespace are not available to source code unless the individual units are explicitly used in the file's uses clause. In other words, if a source file uses only the namespace, then fully qualified identifier expressions referring to a symbol in a unit in that namespace must use the namespace name, not just the name of the unit that defines that symbol.

A uses clause may refer to a namespace as well as individual units within that namespace. In this case, a fully qualified expression referring to a symbol from a specific unit listed in the uses clause may be referred to using the actual unit name or the fully-qualified name (including namespace and unit name) for the qualifier. The two forms of reference are identical and refer to the same symbol.

Note: Explicitly using a unit in the uses clause will only work when you are compiling from source or dcu files. If the namespace units are compiled into an assembly and the assembly is referenced by the project instead of the individual units, then the source code that explicitly refers to a unit in the namespace will fail.

Fundamental Syntactic Elements

This section describes the fundamental syntactic elements, or the building blocks of the Delphi language.

In This Section

[Fundamental Syntactic Elements](#)

Describes the syntax for identifiers, character strings, numbers, and labels.

[Expressions](#)

Describes the syntax of Delphi language expressions.

[Declarations and Statements](#)

Describes the syntax of Delphi declarations and executable statements

Fundamental Syntactic Elements

This topic introduces the Delphi language character set, and describes the syntax for declaring:

- Identifiers
- Numbers
- Character strings
- Labels
- Source code comments

The Delphi Character Set

The Delphi Language uses the Unicode character set, including alphabetic and alphanumeric Unicode characters and the underscore. It is not case-sensitive. The space character and the ASCII control characters (ASCII 0 through 31 including ASCII 13, the return or end-of-line character) are called *blanks*.

The Developer Studio 2006 compiler will accept a file encoded in UCS-2 or UCS-4 if the file contains a byte order mark. The speed of compilation may be penalized by the use for formats other than UTF-8, however. All characters in a UCS-4 encoded source file must be representable in UCS-2 without surrogate pairs. UCS-2 encodings with surrogate pairs (including GB18030) are accepted only if the `codepage` compiler option is specified.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

The Delphi Character Set and Basic Syntax

On the simplest level, a program is a sequence of tokens delimited by separators. A token is the smallest meaningful unit of text in a program. A separator is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;  
Price := 10;
```

Tokens are categorized as special symbols, identifiers, reserved words, directives, numerals, labels, and character strings. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

Special Symbols

Special symbols are non-alphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols:

\$ & ' () * + , - . / : ; < = > @ [] ^ { }

The following character pairs are also special symbols:

(* (. *) .) .. // := <= >= <>

The following table shows equivalent symbols:

Special symbol	Equivalent symbols
[(.
]	.)
{	(*
}	*)

The left bracket [is equivalent to the character pair of left parenthesis and period (.

The right bracket] is equivalent to the character pair of period and right parenthesis .)

The left brace { is equivalent to the character pair of left parenthesis and asterisk (*.

The right brace } is equivalent to the character pair of right parenthesis and asterisk *)

Note: %, ?, \, !, " (double quotation marks), _ (underscore), | (pipe), and ~ (tilde) are not special characters.

Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with an alphabetic character or an underscore (_) and cannot contain spaces; alphanumeric characters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Note: The .NET SDK recommends against using leading underscores in identifiers, as this pattern is reserved for system use.

Since the Delphi Language is case-insensitive, an identifier like `CalculateValue` could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation. For more information, see the topic, *Unit References and the Uses Clause*.

Qualified Identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to qualify the identifier. The syntax for a qualified identifier is

identifier1.identifier2

where *identifier1* qualifies *identifier2*. For example, if two units each declare a variable called `CurrentValue`, you can specify that you want to access the `CurrentValue` in `Unit2` by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the `Click` method in `Button1` of `Form1`.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in Blocks and scope.

Extended Identifiers

Particularly when programming with Delphi for .NET, you might encounter identifiers (e.g. types, or methods in a class) having the same name as a Delphi language keyword. For example, a class might have a method called `begin`. Another example is the CLR class called `Type`, in the `System` namespace. `Type` is a Delphi language keyword, and cannot be used for an identifier name.

If you qualify the identifier with its full namespace specification, then there is no problem. For example, to use the `Type` class, you must use its fully qualified name:

```
var
    TMyType : System.Type; // Using fully qualified namespace
                        // avoids ambiguity with Delphi language keyword.
```

As a shorter alternative, the ampersand (`&`) operator can be used to resolve ambiguities between identifiers and Delphi language keywords. If you encounter a method or type that is the same name as a Delphi keyword, you can omit the namespace specification if you prefix the identifier name with an ampersand. For example, the following code uses the ampersand to disambiguate the CLR `Type` class from the Delphi keyword type

```
var
    TMyType : &Type; // Prefix with '&' is ok.
```

Reserved Words

The following reserved words cannot be redefined or used as identifiers.

Reserved Words

and	else	inherited	packed	then
array	end	initialization	procedure	threadvar
as	except	inline	program	to
asm	exports	interface	property	try
begin	file	is	raise	type
case	final	label	record	unit
class	finalization	library	repeat	unsafe
const	finally	mod	resourcestring	until
constructor	for	nil	sealed	uses
destructor	function	not	set	var
dispinterface	goto	object	shl	while
div	if	of	shr	with
do	implementation	or	static	xor
downto	in	out	string	

In addition to the words above, `private`, `protected`, `public`, `published`, and `automated` act as reserved words within class type declarations, but are otherwise treated as directives. The words `at` and `on` also have special meanings, and should be treated as reserved words.

Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in the Delphi language, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence -- although it is inadvisable to do so -- you can define an identifier that looks exactly like a directive.

Directives

<code>absolute</code>	<code>dynamic</code>	<code>local</code>	<code>platform</code>	<code>requires</code>
<code>abstract</code>	<code>export</code>	<code>message</code>	<code>private</code>	<code>resident</code>
<code>assembler</code>	<code>external</code>	<code>name</code>	<code>protected</code>	<code>safecall</code>
<code>automated</code>	<code>far</code>	<code>near</code>	<code>public</code>	<code>stdcall</code>
<code>cdecl</code>	<code>forward</code>	<code>nodefault</code>	<code>published</code>	<code>stored</code>
<code>contains</code>	<code>implements</code>	<code>overload</code>	<code>read</code>	<code>varargs</code>
<code>default</code>	<code>index</code>	<code>override</code>	<code>readonly</code>	<code>virtual</code>
<code>deprecated</code>	<code>inline</code>	<code>package</code>	<code>register</code>	<code>write</code>
<code>dispid</code>	<code>library</code>	<code>pascal</code>	<code>reintroduce</code>	<code>writeonly</code>

Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the `+` or `-` operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to `+67258`) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character `E` or `e` occurs within a real, it means "times ten to the power of". For example, `7E2` means $7 * 10^2$, and `12.25e+6` and `12.25e6` both mean $12.25 * 10^6$.

The dollar-sign prefix indicates a hexadecimal numeral, for example, `$8F`. Hexadecimal numbers without a preceding `-` unary operator are taken to be positive values. During an assignment, if a hexadecimal value lies outside the range of the receiving type an error is raised, except in the case of the Integer (32-bit integer) where a warning is raised. In this case, values exceeding the positive range for Integer are taken to be negative numbers in a manner consistent with 2's complement integer representation.

For more information about real and integer types, see [Data Types, Variables, and Constants](#). For information about the data types of numerals, see [True constants](#).

Labels

A label is a standard Delphi language identifier with the exception that, unlike other identifiers, labels can start with a digit. Numeric labels can include no more than ten digits - that is, a numeral between 0 and 9999999999.

Labels are used in `goto` statements. For more information about `goto` statements and labels, see [Goto statements](#).

Character Strings

A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND' { BORLAND }
'You'll see' { You'll see }
''         { ' }
''         { null string }
' '        { a space }
```

A control string is a sequence of one or more control characters, each of which consists of the # symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1' #13#10 'Line 2'
```

to put a carriage-returnline-feed between 'Line 1' and 'Line 2'. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the + operator or simply combine them into a single quoted string.)

A character string's length is the number of characters in the string. A character string of any length is compatible with any string type and with the PChar type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (with compiler directive {\$X+}), a nonempty character string of length n is compatible with zero-based arrays and packed arrays of n characters. For more information, see Datatypes, Variables, and Constants.

Comments and Compiler Directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }
(* Text between a left-parenthesis-plus-asterisk and an asterisk-plus-right-
parenthesis is also a comment *)
// Any text between a double-slash and the end of the line constitutes a comment.
```

Comments that are alike cannot be nested. For instance, {{{}} will not work, but (*{}*) will. This is useful for commenting out sections of code that also contain comments.

A comment that contains a dollar sign (\$) immediately after the opening { or (* is a compiler directive. For example,

```
{$WARNINGS OFF}
```

tells the compiler not to generate warning messages.

Declarations and Statements

This topic describes the syntax of Delphi declarations and statements.

Aside from the `uses` clause (and reserved words like `implementation` that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

This topic covers the following items:

- Declarations
- Simple statements such as assignment
- Structured statements such as conditional tests (e.g., `if-then`, and `case`), iteration (e.g., `for`, and `while`).

Declarations

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like 26057 are not identifiers.) Identifiers must be declared before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable `Result` when it occurs inside a function block, and the variable `Self` when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called `Size` that holds an Extended (real) value, while

```
function DoThis(X, Y: string): Integer;
```

declares a function called `DoThis` that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
  Size: Extended;
  Quantity: Integer;
  Description: string;
```

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or implementation section of a unit (after the `uses` clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the documentation for those topics.

Hinting Directives

The 'hint' directives `platform`, `deprecated`, and `library` may be appended to any declaration. These directives will produce warnings at compile time. Hint directives can be applied to type declarations, variable declarations, class, interface and structure declarations, field declarations within classes or records, procedure, function and method declarations, and unit declarations.

When a hint directive appears in a unit declaration, it means that the hint applies to everything in the unit. For example, the Windows 3.1 style `OleAuto.pas` unit on Windows is completely deprecated. Any reference to that unit or any symbol in that unit will produce a deprecation message.

The platform hinting directive on a symbol or unit indicates that it may not exist or that the implementation may vary considerably on different platforms. The library hinting directive on a symbol or unit indicates that the code may not exist or the implementation may vary considerably on different library architectures.

The platform and library directives do not specify which platform or library. If your goal is writing platform-independent code, you do not need to know which platform a symbol is specific to; it is sufficient that the symbol be marked as specific to *some* platform to let you know it may cause problems for your goal of portability.

In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall deprecated;

var
  VersionNumber: Real library;

type
  AppError = class(Exception)
    ...
  end platform;
```

When source code is compiled in the `{ $HINTS ON }` `{ $WARNINGS ON }` state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use platform to mark items that are specific to a particular operating environment (such as Windows or .NET), deprecated to indicate that an item is obsolete or supported only for backward compatibility, and library to flag dependencies on a particular library or component framework.

The Developer Studio 2006 compiler also recognizes the hinting directive experimental. You can use this directive to designate units which are in an unstable, development state. The compiler will emit a warning when it builds an application that uses the unit.

Statements

Statements define algorithmic actions within a program. Simple statements like assignments and procedure calls can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

Simple Statements

A simple statement doesn't contain any other statements. Simple statements include assignments, calls to procedures and functions, and goto jumps.

Assignment Statements

An assignment statement has the form

variable := *expression*

where *variable* is any variable reference, including a variable, variable typecast, dereferenced pointer, or component of a structured variable. The *expression* is any assignment-compatible expression (within a function block, variable can be replaced with the name of the function being defined. See Procedures and functions). The := symbol is sometimes called the assignment operator.

An assignment statement replaces the current value of variable with the value of expression. For example,

```
I := 3;
```

assigns the value 3 to the variable `I`. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

```
I := I + 1;
```

increments the value of `I`. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I * K;
ShortInt(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Procedure and Function Calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X, Y);
```

With extended syntax enabled (`{ $X+ }`), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see [Procedures and functions](#).

Goto Statements

A goto statement, which has the form

```
goto label
```

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

label: statement

Declare labels like this:

```
label label;
```

You can declare several labels at once:

```
label label1, ..., labeln;
```

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and goto statement must belong to the same block. (See Blocks and Scope, below.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;
...
StartHere: Beep;
goto StartHere;
```

creates an infinite loop that calls the `Beep` procedure repeatedly.

Additionally, it is not possible to jump into or out of a try-finally or try-except statement.

The goto statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;
  var X, Y, Z, Count: Integer;
  label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

        ... { Code to execute if no answer is found }
        Exit;

FoundAnAnswer:
  ... { Code to execute when an answer is found }
end;
```

Notice that we are using goto to jump out of a nested loop. Never jump into a loop or other structured statement, since this can have unpredictable effects.

Structured Statements

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or with statement simply executes a sequence of constituent statements.
- A conditional statement that is an if or case statement executes at most one of its constituents, depending on specified criteria.
- Loop statements including repeat, while, and for loops execute a sequence of constituent statements repeatedly.

- A special group of statements including `raise`, `try...except`, and `try...finally` constructions create and handle exceptions. For information about exception generation and handling, see [Exceptions](#).

Compound Statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words `begin` and `end`, and its constituent statements are separated by semicolons. For example:

```
begin
  Z := X;
  X := Y;
  X := Y;
end;
```

The last semicolon before `end` is optional. So we could have written this as

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

Compound statements are essential in contexts where Delphi syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
  I := SomeConstant;
  while I > 0 do
    begin
      ...
      I := I - 1;
    end;
  end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, `begin` and `end` sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

With Statements

A `with` statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a `with` statement is

1 `with obj do statement`, or

2 withobj1, ..., objndostatement

where *obj* is an expression yielding a reference to a record, object instance, class instance, interface or class type (metaclass) instance, and statement is any simple or structured statement. Within the *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone, that is, without qualifiers.

For example, given the declarations

```
type
  TDate = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

var
  OrderDate: TDate;
```

you could write the following with statement.

```
with OrderDate do
  if Month = 12 then
    begin
      Month := 1;
      Year := Year + 1;
    end
  else
    Month := Month + 1;
```

you could write the following with statement.

```
if OrderDate.Month = 12 then
  begin
    OrderDate.Month := 1;
    OrderDate.Year := OrderDate.Year + 1;
  end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before statement is executed. This makes with statements efficient as well as concise. It also means that assignments to a variable within statement cannot affect the interpretation of *obj* during the current execution of the with statement.

Each variable reference or method name in a with statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the with statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year;
    ...
  end;
```


When multiple objects or records appear after with, the entire statement is treated like a series of nested with statements. Thus

withobj1, obj2, ..., objndostatement

is equivalent to

```
with obj1 do
  with obj2 do
    ...
    with objn do
      // statement
```

In this case, each variable reference or method name in *statement* is interpreted, if possible, as a member of *objn*; otherwise it is interpreted, if possible, as a member of *objn1*; and so forth. The same rule applies to interpreting the *objs* themselves, so that, for instance, if *objn* is a member of both *obj1* and *obj2*, it is interpreted as *obj2.objn*.

If Statements

There are two forms of if statement: if...then and the if...then...else. The syntax of an if...then statement is

ifexpressionthenstatement

where *expression* returns a Boolean value. If *expression* is True, then *statement* is executed; otherwise it is not. For example,

```
if J <> 0 then Result := I / J;
```

The syntax of an if...then...else statement is

ifexpressionthenstatement1elsestatement2

where *expression* returns a Boolean value. If *expression* is True, then *statement1* is executed; otherwise *statement2* is executed. For example,

```
if J = 0 then
  Exit
else
  Result := I / J;
```

The then and else clauses contain one statement each, but it can be a structured statement. For example,

```
if J <> 0 then
  begin
    Result := I / J;
    Count := Count + 1;
  end
else if Count = Last then
  Done := True
else
  Exit;
```

Notice that there is never a semicolon between the then clause and the word else. You can place a semicolon after an entire if statement to separate it from the next statement in its block, but the then and else clauses require nothing

more than a space or carriage return between them. Placing a semicolon immediately before else (in an if statement) is a common programming error.

A special difficulty arises in connection with nested if statements. The problem arises because some if statements have else clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer else clauses than if statements, it may not seem clear which else clauses are bound to which ifs. Consider a statement of the form

ifexpression1thenifexpression2thenstatement1elsestatement2;

There would appear to be two ways to parse this:

ifexpression1 then [ifexpression2thenstatement1elsestatement2];

ifexpression1then [ifexpression2thenstatement1] elsestatement2;

The compiler always parses in the first way. That is, in real code, the statement

```
if ... { expression1 } then
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 }
```

is equivalent to

```
if ... { expression1 } then
  begin
    if ... { expression2 } then
      ... { statement1 }
    else
      ... { statement2 }
  end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each else bound to the nearest available if on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... { expression1 } then
  begin
    if ... { expression2 } then
      ... { statement1 }
    end
  end
else
  ... { statement2 };
```

Case Statements

The case statement may provide a readable alternative to deeply nested if conditionals. A case statement has the form

```
case selectorExpression of
  caseList1: statement1;
```

```

    ...
    caseListn: statementn;
end

```

where *selectorExpression* is any expression of an ordinal type smaller than 32 bits (string types and ordinals larger than 32 bits are invalid) and each *caseList* is one of the following:

- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus 7, True, 4 + 5 * 3, 'A', and Integer('A') can all be used as *caseLists*, but variables and most function calls cannot. (A few built-in functions like Hi and Lo can occur in a *caseList*. See Constant expressions.)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item1*, ..., *itemn*, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the case statement; subranges and lists cannot overlap. A case statement can have a final else clause:

```

case selectorExpression of
    caseList1: statement1;
    ...
    caselistn: statementn;
else
    statements;
end

```

where *statements* is a semicolon-delimited sequence of statements. When a case statement is executed, at most one of *statement1* ... *statementn* is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the statement to be used. If none of the *caseLists* has the same value as *selectorExpression*, then the statements in the else clause (if there is one) are executed.

The case statement

```

case I of
    1..5: Caption := 'Low';
    6..9: Caption := 'High';
    0, 10..99: Caption := 'Out of range';
else
    Caption := '';
end

```

is equivalent to the nested conditional

```

if I in [1..5] then
    Caption := 'Low';
else if I in [6..10] then
    Caption := 'High';
else if (I = 0) or (I in [10..99]) then
    Caption := 'Out of range'
else
    Caption := '';

```

Other examples of case statements

```

case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X := 3;
  Yellow, Orange, Black: X := 0;
end;

case Selection of
  Done: Form1.Close;
  Compute: calculateTotal(UnitCost, Quantity);
  else
    Beep;
end;

```

Control Loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Delphi has three kinds of control loop: repeat statements, while statements, and for statements.

You can use the standard `Break` and `Continue` procedures to control the flow of a repeat, while, or for statement. `Break` terminates the statement in which it occurs, while `Continue` begins executing the next iteration of the sequence.

Repeat Statements

The syntax of a **repeat** statement is

repeatstatement1; ...; statementn;untilexpression

where *expression* returns a Boolean value. (The last semicolon before *until* is optional.) The repeat statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns True, the repeat statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of repeat statements include

```

repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);

```

While Statements

A while statement is similar to a repeat statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a while statement is

whileexpressiondo statement

where *expression* returns a Boolean value and *statement* can be a compound statement. The while statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns True, execution continues.

Examples of while statements include

```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

For Statements

A for statement, unlike a repeat or while statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a for statement is

*for*counter := *initialValue* to *finalValue* do *statement*

or

*for*counter := *initialValue* down to *finalValue* do *statement*

where

- *counter* is a local variable (declared in the block containing the for statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with counter.
- *statement* is a simple or structured statement that does not change the value of counter.

The for statement assigns the value of *initialValue* to *counter*, then executes *statement* repeatedly, incrementing or decrementing *counter* after each iteration. (The for...to syntax increments *counter*, while the for...downto syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the for statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a for...to statement, or less than *finalValue* in a for...downto statement, then *statement* is never executed. After the for statement terminates (provided this was not forced by a [Break](#) or an [Exit](#) procedure), the value of *counter* is undefined.

For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the for...to statement is almost, but not quite, equivalent to this while construction:

```
begin
  counter := initialValue;
  while counter <= finalValue do
  begin
    ... {statement};
```

```

        counter := Succ(counter);
    end;
end

```

The difference between this construction and the `for...to` statement is that the while loop reevaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of `for` statements:

```

for I := 2 to 63 do
    if Data[I] > Max then
        Max := Data[I];
    end;
end;

for I := ListBox1.Items.Count - 1 downto 0 do
    ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
end;

for I := 1 to 10 do
    for J := 1 to 10 do
        begin
            X := 0;
            for K := 1 to 10 do
                X := X + Mat1[I,K] * Mat2[K,J];
            end;
            Mat[I,J] := X;
        end;
    end;
end;

for C := Red to Blue do Check(C);
end;

```

Iteration Over Containers Using For statements

Both Delphi for .NET and for Win32 support `for-element-in-collection` style iteration over containers. The following container iteration patterns are recognized by the compiler:

- `for Element in ArrayExpr do Stmt;`
- `for Element in StringExpr do Stmt;`
- `for Element in SetExpr do Stmt;`
- `for Element in CollectionExpr do Stmt;`

The type of the iteration variable `Element` must match the type held in the container. With each iteration of the loop, the iteration variable holds the current collection member. As with regular `for`-loops, the iteration variable must be declared within the same block as the `for` statement. The iteration variable cannot be modified within the loop. This includes assignment, and passing the variable to a `var` parameter of a procedure. Doing so will result in a compile-time error.

Array expressions can be single or multidimensional, fixed length, or dynamic arrays. The array is traversed in increasing order, starting at the lowest array bound and ending at the array size minus one. The following code shows an example of traversing single, multi-dimensional, and dynamic arrays:

```

type
    TIntArray      = array[0..9] of Integer;
    TGenericIntArray = array of Integer;

var
    IArray1: array[0..9] of Integer = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    IArray2: array[1..10] of Integer = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

```

```

IArray3: array[1..2] of TIntArray = ((11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
                                     (21, 22, 23, 24, 25, 26, 27, 28, 29, 30));

MultiDimTemp: TIntArray;
IDynArray:    TGenericIntArray;

I: Integer;

begin

  for I in IArray1 do
    begin
      // Do something with I...
    end;

  // Indexing begins at lower array bound of 1.
  for I in IArray2 do
    begin
      // Do something with I...
    end;

  // Iterating a multi-dimensional array
  for MultiDimTemp in IArray3 do // Indexing from 1..2
    for I in MultiDimTemp do     // Indexing from 0..9
      begin
        // Do something with I...
      end;

  // Iterating over a dynamic array
  IDynArray := IArray1;
  for I in IDynArray do
    begin
      // Do something with I...
    end;

```

The following example demonstrates iteration over string expressions:

```

var
  C: Char;
  S1, S2: String;
  Counter: Integer;

  OS1, OS2: ShortString;
  AC: AnsiChar;

begin

  S1 := 'Now is the time for all good men to come to the aid of their country.';
  S2 := '';

  for C in S1 do
    S2 := S2 + C;

  if S1 = S2 then
    WriteLn('SUCCESS #1');
  else
    WriteLn('FAIL #1');

  OS1 := 'When in the course of human events it becomes necessary to dissolve...';
  OS2 := '';

```

```

for AC in OS1 do
    OS2 := OS2 + AC;

if OS1 = OS2 then
    WriteLn('SUCCESS #2');
else
    WriteLn('FAIL #2');

end.

```

The following example demonstrates iteration over set expressions:

```

type

    TMyThing = (one, two, three);
    TMySet    = set of TMyThing;
    TCharSet  = set of Char;

var
    MySet:    TMySet;
    MyThing: TMyThing;

    CharSet: TCharSet;
    {$IF DEFINED(CLR)}
    C: AnsiChar;
    {$ELSE}
    C: Char;
    {$IFEND}

begin

    MySet := [one, two, three];
    for MyThing in MySet do
        begin
            // Do something with MyThing...
        end;

    CharSet := [#0..#255];
    for C in CharSet do
        begin
            // Do something with C...
        end;

end.

```

To use the for-in loop construct on a class, the class must implement a prescribed collection pattern. A type that implements the collection pattern must have the following attributes:

- The class must contain a public instance method called `GetEnumerator()`. The `GetEnumerator()` method must return a class, interface, or record type.
- The class, interface, or record returned by `GetEnumerator()` must contain a public instance method called `MoveNext()`. The `MoveNext()` method must return a Boolean.
- The class, interface, or record returned by `GetEnumerator()` must contain a public instance, read-only property called `Current`. The type of the `Current` property must be the type contained in the collection.

If the enumerator type returned by `GetEnumerator()` implements the `IDisposable` interface, the compiler will call the `Dispose` method of the type when the loop terminates.

The following code demonstrates iterating over an enumerable container in Delphi.

```
type
  TMyIntArray = array of Integer;

  TMyEnumerator = class
    Values: TMyIntArray;
    Index: Integer;
  public
    constructor Create;
    function GetCurrent: Integer;
    function MoveNext: Boolean;
    property Current: Integer read GetCurrent;
  end;

  TMyContainer = class
  public
    function GetEnumerator: TMyEnumerator;
  end;

constructor TMyEnumerator.Create;
begin
  inherited Create;
  Values := TMyIntArray.Create(100, 200, 300);
  Index := -1;
end;

function TMyEnumerator.MoveNext: Boolean;
begin
  if Index < High(Values) then
  begin
    Inc(Index);
    Result := True;
  end
  else
    Result := False;
end;

function TMyEnumerator.GetCurrent: Integer;
begin
  Result := Values[Index];
end;

function TMyContainer.GetEnumerator: TMyEnumerator;
begin
  Result := TMyEnumerator.Create;
end;

var
  MyContainer: TMyContainer;
  I: Integer;

  Counter: Integer;

begin
  MyContainer := TMyContainer.Create;
```

```
Counter := 0;
for I in MyContainer do
    Inc(Counter, I);

WriteLn('Counter = ', Counter);
end.
```

The following classes and their descendents support the for-in syntax:

- TList
- TCollection
- TStrings
- TInterfaceList
- TComponent
- TMenuItem
- TCustomActionList
- TFields
- TListItems
- TTreeNode
- TToolBar

Blocks and Scope

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

Blocks

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```
{declarations}
begin
    {statements}
end
```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program block, the *declarations* section can also include one or more exports clauses (see Libraries and packages).

For example, in a function declaration like

```
function UpperCase(const S: string): string;
var
    Ch: Char;
    L: Integer;
    Source, Dest: PChar;
begin
```

```
...
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. `Ch`, `L`, `Source`, and `Dest` are local variables; their declarations apply only to the `UpperCase` function block and override, in this block only, any declarations of the same identifiers that may occur in the program block or in the interface or implementation section of a unit.

Scope

An identifier, such as a variable or function name, can be used only within the scope of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope, especially identifiers declared in functions and procedures, are sometimes called local, while identifiers with wider scope are called global.

The rules that determine identifier scope are summarized below.

If the identifier is declared in ...	its scope extends ...
the declaration section of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Programs and Units.)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See Records.)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Classes and Objects.)

Naming Conflicts

When one block encloses another, the former is called the outer block and the latter the inner block. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration takes precedence over the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called `MaxValue` in the interface section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of `MaxValue` in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a `uses` clause imposes a new scope that encloses the remaining units used and the program or unit containing the `uses` clause. The first unit in a `uses` clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their interface sections, an unqualified reference to the identifier selects the declaration in the innermost scope, that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the `uses` clause that does declare the identifier.

The `System` and `SysInit` units are used automatically by every program or unit. The declarations in `System`, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and bypass an inner declaration by using a qualified identifier (see Qualified Identifiers) or a with statement (see With Statements, above).

Expressions

This topic describes syntax rules of forming Delphi expressions.

The following items are covered in this topic:

- Valid Delphi Expressions
- Operators
- Function calls
- Set constructors
- Indexes
- Typecasts

Expressions

An expression is a construction that returns a value. The following table shows examples of Delphi expressions:

<code>X</code>	variable
<code>@X</code>	address of the variable <code>X</code>
<code>15</code>	integer constant
<code>InterestRate</code>	variable
<code>Calc(X, Y)</code>	function call
<code>X * Y</code>	product of <code>X</code> and <code>Y</code>
<code>Z / (1 - Z)</code>	quotient of <code>Z</code> and <code>(1 - Z)</code>
<code>X = 1.5</code>	Boolean
<code>C in Range1</code>	Boolean
<code>not Done</code>	negation of a Boolean
<code>['a', 'b', 'c']</code>	set
<code>Char(48)</code>	value typecast

The simplest expressions are variables and constants (described in Data types, variables, and constants). More complex expressions are built from simpler ones using operators, function calls, set constructors, indexes, and typecasts.

Operators

Operators behave like predefined functions that are part of the Delphi language. For example, the expression `(X + Y)` is built from the variables `X` and `Y`, called operands, with the `+` operator; when `X` and `Y` represent integers or reals, `(X + Y)` returns their sum. Operators include `@`, `not`, `^`, `*`, `/`, `div`, `mod`, `and`, `shl`, `shr`, `as`, `+`, `-`, `or`, `xor`, `=`, `>`, `<`, `<>`, `<=`, `>=`, `in`, and `is`.

The operators `@`, `not`, and `^` are unary (taking one operand). All other operators are binary (taking two operands), except that `+` and `-` can function as either a unary or binary operator. A unary operator always precedes its operand (for example, `-B`), except for `^`, which follows its operand (for example, `P^`). A binary operator is placed between its operands (for example, `A = 7`).

Some operators behave differently depending on the type of data passed to them. For example, not performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories.

Except for `^`, `is`, and `in`, all operators can take operands of type Variant.

The sections that follow assume some familiarity with Delphi data types.

For information about operator precedence in complex expressions, see Operator Precedence Rules, later in this topic.

Arithmetic Operators

Arithmetic operators, which take real or integer operands, include `+`, `-`, `*`, `/`, `div`, and `mod`.

Binary Arithmetic Operators

Operator	Operation	Operand Types	Result Type	Example
<code>+</code>	addition	integer, real	integer, real	<code>X + Y</code>
<code>-</code>	subtraction	integer, real	integer, real	<code>Result - 1</code>
<code>*</code>	multiplication	integer, real	integer, real	<code>P * InterestRate</code>
<code>/</code>	real division	integer, real	real	<code>X / 2</code>
<code>div</code>	integer division	integer	integer	<code>Total div UnitSize</code>
<code>mod</code>	remainder	integer	integer	<code>Y mod 6</code>

Unary arithmetic operators

Operator	Operation	Operand Type	Result Type	Example
<code>+</code>	sign identity	integer, real	integer, real	<code>+7</code>
<code>-</code>	sign negation	integer, real	integer, real	<code>-X</code>

The following rules apply to arithmetic operators.

- The value of `x / y` is of type Extended, regardless of the types of `x` and `y`. For other arithmetic operators, the result is of type Extended whenever at least one operand is a real; otherwise, the result is of type Int64 when at least one operand is of type Int64; otherwise, the result is of type Integer. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of `x div y` is the value of `x / y` rounded in the direction of zero to the nearest integer.
- The mod operator returns the remainder obtained by dividing its operands. In other words, `x mod y = x - (x div y) * y`.
- A runtime error occurs when `y` is zero in an expression of the form `x / y`, `x div y`, or `x mod y`.

Boolean Operators

The Boolean operators not, and, or, and xor take operands of any Boolean type and return a value of type Boolean.

Boolean Operators

Operator	Operation	Operand Types	Result Type	Example
<code>not</code>	negation	Boolean	Boolean	<code>not (C in MySet)</code>
<code>and</code>	conjunction	Boolean	Boolean	<code>Done and (Total > 0)</code>
<code>or</code>	disjunction	Boolean	Boolean	<code>A or B</code>

xor	exclusive disjunction	Boolean	Boolean	<code>A xor B</code>
-----	-----------------------	---------	---------	----------------------

These operations are governed by standard rules of Boolean logic. For example, an expression of the form `x and y` is True if and only if both `x` and `y` are True.

Complete Versus Short-Circuit Boolean Evaluation

The compiler supports two modes of evaluation for the `and` and `or` operators: complete evaluation and short-circuit (partial) evaluation. Complete evaluation means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. Short-circuit evaluation means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when `A` is False, the compiler won't evaluate `B`; it knows that the entire expression is False as soon as it evaluates `A`.

Short-circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string `S`, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
    ...
    Inc(I);
end;
```

In the case where `S` has no commas, the last iteration increments `I` to a value which is greater than the length of `S`. When the while condition is next tested, complete evaluation results in an attempt to read `S[I]`, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the while condition (`S[I] <> ', '`) is not evaluated after the first part fails.

Use the `$B` compiler directive to control evaluation mode. The default state is `{ $B }`, which enables short-circuit evaluation. To enable complete evaluation locally, add the `{ $B+ }` directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting **Complete Boolean Evaluation** in the **Compiler Options** dialog (all source units will need to be recompiled).

Note: If either operand involves a Variant, the compiler always performs complete evaluation (even in the `{ $B }` state).

Logical (Bitwise) Operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in `x` (in binary) is `001101` and the value stored in `y` is `100001`, the statement

```
Z := X or Y;
```

assigns the value `101101` to `Z`.

Logical (Bitwise) Operators

Operator	Operation	Operand Types	Result Type	Example
not	bitwise negation	integer	integer	<code>not X</code>
and	bitwise and	integer	integer	<code>X and Y</code>

or	bitwise or	integer	integer	<code>X or Y</code>
xor	bitwise xor	integer	integer	<code>X xor Y</code>
shl	bitwise shift left	integer	integer	<code>X shl 2</code>
shr	bitwise shift right	integer	integer	<code>Y shr I</code>

The following rules apply to bitwise operators.

- The result of a not operation is of the same type as the operand.
- If the operands of an and, or, or xor operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations `x shl y` and `x shr y` shift the value of `x` to the left or right by `y` bits, which (if `x` is an unsigned integer) is equivalent to multiplying or dividing `x` by 2^y ; the result is of the same type as `x`. For example, if `N` stores the value `01101` (decimal 13), then `N sh 1` returns `11010` (decimal 26). Note that the value of `y` is interpreted modulo the size of the type of `x`. Thus for example, if `x` is an integer, `x shl 40` is interpreted as `x shl 8` because an integer is 32 bits and $40 \bmod 32$ is 8.

String Operators

The relational operators `=`, `<>`, `<`, `>`, `<=`, and `>=` all take string operands (see Relational operators). The `+` operator concatenates two strings.

String Operators

Operator	Operation	Operand Types	Result Type	Example
+	concatenation	string, packed string, character	string	<code>S + ' . '</code>

The following rules apply to string concatenation.

- The operands for `+` can be strings, packed strings (packed arrays of type `Char`), or characters. However, if one operand is of type `WideChar`, the other operand must be a long string (`AnsiString` or `WideString`).
- The result of a `+` operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

Pointer Operators

The relational operators `<`, `>`, `<=`, and `>=` can take operands of type `PChar` and `PWideChar` (see Relational operators). The following operators also take pointers as operands. For more information about pointers, see Pointers and pointer types.

Character-pointer operators

Operator	Operation	Operand Types	Result Type	Example
+	pointer addition	character pointer, integer	character pointer	<code>P + I</code>
-	pointer subtraction	character pointer, integer	character pointer, integer	<code>P - Q</code>
^	pointer dereference	pointer	base type of pointer	<code>P^</code>
=	equality	pointer	Boolean	<code>P = Q</code>
<>	inequality	pointer	Boolean	<code>P <> Q</code>

The `^` operator dereferences a pointer. Its operand can be a pointer of any type except the generic `Pointer`, which must be typecast before dereferencing.

$P = Q$ is True just in case P and Q point to the same address; otherwise, $P <> Q$ is True.

You can use the $+$ and $-$ operators to increment and decrement the offset of a character pointer. You can also use $-$ to calculate the difference between the offsets of two character pointers. The following rules apply.

- If I is an integer and P is a character pointer, then $P + I$ adds I to the address given by P ; that is, it returns a pointer to the address I characters after P . (The expression $I + P$ is equivalent to $P + I$.) $P - I$ subtracts I from the address given by P ; that is, it returns a pointer to the address I characters before P . This is true for PChar pointers; for PWideChar pointers $P + I$ adds `SizeOf(WideChar)` to P .
- If P and Q are both character pointers, then $P - Q$ computes the difference between the address given by P (the higher address) and the address given by Q (the lower address); that is, it returns an integer denoting the number of characters between P and Q . $P + Q$ is not defined.

Set Operators

The following operators take sets as operands.

Set Operators

Operator	Operation	Operand Types	Result Type	Example
$+$	union	set	set	$Set1 + Set2$
$-$	difference	set	set	$S - T$
$*$	intersection	set	set	$S * T$
\leq	subset	set	Boolean	$Q \leq MySet$
\geq	superset	set	Boolean	$S1 \geq S2$
$=$	equality	set	Boolean	$S2 = MySet$
$<>$	inequality	set	Boolean	$MySet <> S1$
in	membership	ordinal, set	Boolean	$A \text{ in } Set1$

The following rules apply to $+$, $-$, and $*$.

- An ordinal O is in $X + Y$ if and only if O is in X or Y (or both). O is in $X - Y$ if and only if O is in X but not in Y . O is in $X * Y$ if and only if O is in both X and Y .
- The result of a $+$, $-$, or $*$ operation is of the type `set of A..B`, where A is the smallest ordinal value in the result set and B is the largest.

The following rules apply to \leq , \geq , $=$, $<>$, and **in**.

- $X \leq Y$ is True just in case every member of X is a member of Y ; $Z \geq W$ is equivalent to $W \leq Z$. $U = V$ is True just in case U and V contain exactly the same members; otherwise, $U <> V$ is True.
- For an ordinal O and a set S , $O \text{ in } S$ is True just in case O is a member of S .

Relational Operators

Relational operators are used to compare two operands. The operators $=$, $<>$, \leq , and \geq also apply to sets.

Relational Operators

Operator	Operation	Operand Types	Result Type	Example
$=$	equality	simple, class, class reference, interface, string, packed string	Boolean	$I = Max$

<>	inequality	simple, class, class reference, interface, string, packed string	Boolean	<code>X <> Y</code>
<	less-than	simple, string, packed string, PChar	Boolean	<code>X < Y</code>
>	greater-than	simple, string, packed string, PChar	Boolean	<code>Len > 0</code>
<=	less-than-or-equal-to	simple, string, packed string, PChar	Boolean	<code>Cnt <= I</code>
>=	greater-than-or-equal-to	simple, string, packed string, PChar	Boolean	<code>I >= 1</code>

For most simple types, comparison is straightforward. For example, `I = J` is True just in case `I` and `J` have the same value, and `I <> J` is True otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordinal values that make up the characters that make up the string. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with `n` components is compared to a string, the packed string is treated as a string of length `n`.
- Use the operators `<`, `>`, `<=`, and `>=` to compare PChar (and PWideChar) operands only if the two pointers point within the same character array.
- The operators `=` and `<>` can take operands of class and class-reference types. With operands of a class type, `=` and `<>` are evaluated according the rules that apply to pointers: `C = D` is True just in case `C` and `D` point to the same instance object, and `C <> D` is True otherwise. With operands of a class-reference type, `C = D` is True just in case `C` and `D` denote the same class, and `C <> D` is True otherwise. This does not compare the data stored in the classes. For more information about classes, see [Classes and objects](#).

Class Operators

The operators `as` and `is` take classes and instance objects as operands; `as` operates on interfaces as well. For more information, see [Classes and objects](#) and [Object interfaces](#).

The relational operators `=` and `<>` also operate on classes.

The @ Operator

The `@` operator returns the address of a variable, or of a function, procedure, or method; that is, `@` constructs a pointer to its operand. For more information about pointers, see [Pointers and pointer types](#). The following rules apply to `@`.

- If `X` is a variable, `@X` returns the address of `X`. (Special rules apply when `X` is a procedural variable; see [Procedural types in statements and expressions](#).) The type of `@X` is `Pointer` if the default `{ $T }` compiler directive is in effect. In the `{ $T+ }` state, `@X` is of type `^T`, where `T` is the type of `X` (this distinction is important for assignment compatibility, see [Assignment-compatibility](#)).
- If `F` is a routine (a function or procedure), `@F` returns `F`'s entry point. The type of `@F` is always `Pointer`.
- When `@` is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

```
@TMyClass.DoSomething
```

points to the `DoSomething` method of `TMyClass`. For more information about classes and methods, see [Classes and objects](#).

Note: When using the @ operator, it is not possible to take the address of an interface method as the address is not known at compile time and cannot be extracted at runtime.

Operator Precedence

In complex expressions, rules of precedence determine the order in which operations are performed.

Precedence of operators

Operators	Precedence
@, not	first (highest)
*, /, div, mod, and, shl, shr, as	second
+, -, or, xor	third
=, <>, <, >, <=, >=, in, is	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

```
X + Y * Z
```

multiplies `Y` times `Z`, then adds `X` to the result; `*` is performed first, because it has a higher precedence than `+`. But

```
X - Y + Z
```

first subtracts `Y` from `X`, then adds `Z` to the result; `-` and `+` have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

```
(X + Y) * Z
```

multiplies `Z` times the sum of `X` and `Y`.

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

```
X = Y or X = Z
```

The intended interpretation of this is obviously

```
(X = Y) or (X = Z)
```

Without parentheses, however, the compiler follows operator precedence rules and reads it as

```
(X = (Y or X)) = Z
```

which results in a compilation error unless `Z` is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example could be written as

```
X + (Y * Z)
```

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

Function Calls

Because functions return a value, function calls are expressions. For example, if you've defined a function called `Calc` that takes two integer arguments and returns an integer, then the function call `Calc(24,47)` is an integer expression. If `I` and `J` are integer variables, then `I + Calc(J,8)` is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

For more information about functions, see Procedures and functions.

Set Constructors

A set constructor denotes a set-type value. For example,

```
[5, 6, 7, 8]
```

denotes the set whose members are 5, 6, 7, and 8. The set constructor

```
[ 5..8 ]
```

could also denote the same set.

The syntax for a set constructor is

[item1, ..., itemn]

where each item is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (..) in between. When an item has the form `x..y`, it is shorthand for all the ordinals in the range from `x` to `y`, including `y`; but if `x` is greater than `y`, then `x..y`, the set `[x..y]`, denotes nothing and is the empty set. The set constructor `[]` denotes the empty set, while `[x]` denotes the set whose only member is the value of `x`.

Examples of set constructors:

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see Sets.

Indexes

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if `FileName` is a string variable, the expression `FileName[3]` returns the third character in the string denoted by `FileName`, while `FileName[I + 1]` returns the character immediately after the one indexed by `I`. For information about strings, see String types. For information about arrays and array properties, see Arrays and Array properties.

Typecasts

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, `Integer('A')` casts the character `A` as an integer.

The syntax for a typecast is

typeIdentifier(expression)

If the expression is a variable, the result is called a variable typecast; otherwise, the result is a value typecast. While their syntax is the same, different rules apply to the two kinds of typecast.

Value Typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal or pointer types. Examples of value typecasts include

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

The statement

```
I := Integer('A');
```

assigns the value of `Integer('A')`, which is 65, to the variable `I`.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

Variable Typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like `Int` and `Trunc`.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
...
Shortint(MyChar) := 122;
```

assigns the character `z` (ASCII 122) to `MyChar`.

You can cast variables to a procedural type. For example, given the declarations

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can make the following assignments.

```
F := Func(P);      { Assign procedural value in P to F }
Func(P) := F;      { Assign procedural value in F to P }
@F := P;           { Assign pointer value in P to F }
P := @F;           { Assign pointer value in F to P }
N := F(N);          { Call function via F }
N := Func(P)(N);   { Call function via P }
```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;

  TWordRec = record
    Low, High: Word;
  end;

var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;

begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $1234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

In this example, `TByteRec` is used to access the low- and high-order bytes of a word, and `TWordRec` to access the low- and high-order words of a long integer. You could call the predefined functions `Lo` and `Hi` for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see [Pointers and pointer types](#). For information about casting class and interface types, see [The as operator](#) and [Interface typecasts](#).

Data Types, Variables, and Constants

This section describes the fundamental data types of the Delphi language.

In This Section

[Data Types, Variables, and Constants](#)

A conceptual overview of data types in the Delphi language.

[Simple Types](#)

Describes ordinal and real data types in the Delphi language.

[String Types](#)

Describes the string data types available in the Delphi language.

[Structured Types](#)

Describes how to declare and use structured data types such as arrays and records.

[Pointers and Pointer Types](#)

Describes the syntax of pointer data types in the Delphi language.

[Procedural Types](#)

Describes the use of procedural data types in Delphi.

[Variant Types](#)

Describes the use of variant types in Delphi.

[Type Compatibility and Identity](#)

Describes data type compatibility in Delphi.

[Declaring Types](#)

Describes the syntax of Delphi type declarations.

[Variables](#)

Describes the syntax of Delphi variable declarations.

[Declared Constants](#)

Describes the syntax of constant declarations.

Data Types, Variables, and Constants

This topic presents a high-level overview of Delphi data types.

About Types

A type is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

The Delphi language is a 'strongly typed' language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include typecasting, pointers, Variants, Variant parts in records, and absolute addressing of variables.

There are several ways to categorize Delphi data types:

- Some types are predefined (or built-in); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either fundamental or generic. The range and format of a fundamental type is the same in all implementations of the Delphi language, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It's a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage format from one implementation of a generic type to the next could cause compatibility problems - for example, if you are streaming content to a file as raw, binary data, without type and versioning information.
- Types can be classified as simple, string, structured, pointer, procedural, or variant. In addition, type identifiers themselves can be regarded as belonging to a special 'type' because they can be passed as parameters to certain functions (such as High, Low, and SizeOf).

The outline below shows the taxonomy of Delphi data types.

```
simple
  ordinal
    integer
    character
    Boolean
    enumerated
    subrange
  real
string
structured
  set
  array
  record
  file
  class
  class reference
  interface
pointer
procedural
```

```
Variant  
type identifier
```

The standard function `SizeOf` operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) required to store data of the specified type. For example, `SizeOf(Longint)` returns 4, since a Longint variable uses four bytes of memory.

Type declarations are illustrated in the topics that follow. For general information about type declarations, see [Declaring types](#).

Simple Types

Simple types - which include ordinal types and real types - define ordered sets of values.

The ordinal types covered in this topic are:

- Integer types
- Character types
- Boolean types
- Enumerated types
- Real (floating point) types

Ordinal Types

Ordinal types include integer, character, Boolean, enumerated, and subrange types. An ordinal type defines an ordered set of values in which each value except the first has a unique predecessor and each value except the last has a unique successor. Further, each value has an ordinality which determines the ordering of the type. In most cases, if a value has ordinality n , its predecessor has ordinality $n-1$ and its successor has ordinality $n+1$.

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
<code>Ord</code>	ordinal expression	ordinality of expression's value	Does not take Int64 arguments.
<code>Pred</code>	ordinal expression	predecessor of expression's value	
<code>Succ</code>	ordinal expression	successor of expression's value	
<code>High</code>	ordinal type identifier or variable of ordinal type	highest value in type	Also operates on short-string types and arrays.
<code>Low</code>	ordinal type identifier or variable of ordinal type	lowest value in type	Also operates on short-string types and arrays.

For example, `High(Byte)` returns 255 because the highest value of type Byte is 255, and `Succ(2)` returns 3 because 3 is the successor of 2.

The standard procedures `Inc` and `Dec` increment and decrement the value of an ordinal variable. For example, `Inc(I)` is equivalent to `I := Succ(I)` and, if `I` is an integer variable, to `I := I + 1`.

Integer Types

An integer type represents a subset of the whole numbers. The generic integer types are Integer and Cardinal; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the Delphi compiler.

Generic integer types

Type	Range	Format	.NET Type Mapping
Integer	-2147483648..2147483647	signed 32-bit	Int32
Cardinal	0..4294967295	unsigned 32-bit	UInt32

Fundamental integer types include Shortint, Smallint, Longint, Int64, Byte, Word, and Longword.

Fundamental integer types

Type	Range	Format	.NET Type Mapping
Shortint	-128..127	signed 8-bit	SByte
Smallint	-32768..32767	signed 16-bit	Int16
Longint	-2147483648..2147483647	signed 32-bit	Int32
Int64	$-2^{63}..2^{63}-1$	signed 64-bit	Int64
Byte	0..255	unsigned 8-bit	Byte
Word	0..65535	unsigned 16-bit	UInt16
Longword	0..4294967295	unsigned 32-bit	UInt32

In general, arithmetic operations on integers return a value of type Integer, which is equivalent to the 32-bit Longint. Operations return a value of type Int64 only when performed on one or more Int64 operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
  ...

  I := High(Integer);
  J := I + 1;
```

To get an Int64 return value in this situation, cast `I` as Int64:

```
...
J := Int64(I) + 1;
```

For more information, see Arithmetic operators.

Note: Some standard routines that take integer arguments truncate Int64 values to 32 bits. However, the [High](#), [Low](#), [Succ](#), [Pred](#), [Inc](#), [Dec](#), [IntToStr](#), and [IntToHex](#) routines fully support Int64 arguments. Also, the [Round](#), [Trunc](#), [StrToInt64](#), and [StrToInt64Def](#) functions return Int64 values. A few routines cannot take Int64 values at all.

When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the Shortint type has the range 128..127; hence, after execution of the code

```
var I: Shortint;
...
I := High(Shortint);
I := I + 1;
```

the value of `I` is 128. If compiler range-checking is enabled, however, this code generates a runtime error.

Character Types

The fundamental character types are `AnsiChar` and `WideChar`. `AnsiChar` values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. `AnsiChar` was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

`WideChar` characters use more than one byte to represent every character. In the current implementations, `WideChar` is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is `Char`, which is equivalent to `AnsiChar` on Win32, and to `Char` on the .NET platform. Because the implementation of `Char` is subject to change, it's a good idea to use the standard function `SizeOf` rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

Note: The `WideChar` type also maps to `Char` on the .NET platform.

A string constant of length 1, such as 'A', can denote a character value. The predefined function `Chr` returns the character value for any integer in the range of `AnsiChar` or `WideChar`; for example, `Chr(65)` returns the letter A.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

```
var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
  end;
```

`Letter` has the value A (ASCII 65).

Boolean Types

The four predefined Boolean types are `Boolean`, `ByteBool`, `WordBool`, and `LongBool`. `Boolean` is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A `Boolean` variable occupies one byte of memory, a `ByteBool` variable also occupies one byte, a `WordBool` variable occupies two bytes (one word), and a `LongBool` variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants `True` and `False`. The following relationships hold.

Boolean	ByteBool, WordBool, LongBool
<i>False</i> < <i>True</i>	<i>False</i> <> <i>True</i>
<i>Ord(False)</i> = 0	<i>Ord(False)</i> = 0
<i>Ord(True)</i> = 1	<i>Ord(True)</i> <> 0
<i>Succ(False)</i> = <i>True</i>	<i>Succ(False)</i> = <i>True</i>
<i>Pred(True)</i> = <i>False</i>	<i>Pred(False)</i> = <i>True</i>

A value of type `ByteBool`, `LongBool`, or `WordBool` is considered `True` when its ordinality is nonzero. If such a value appears in a context where a `Boolean` is expected, the compiler automatically converts any value of nonzero ordinality to `True`.

The previous remarks refer to the ordinality of Boolean values, not to the values themselves. In Delphi, Boolean expressions cannot be equated with integers or reals. Hence, if X is an integer variable, the statement

```
if X then ...;
```

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```
if X <> 0 then ...; { use longer expression that returns Boolean value }

var OK: Boolean;
...
if X <> 0 then OK := True;
if OK then ...;
```

Enumerated Types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

`typeName = (val1 , ..., valn)`

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called `Suit` whose possible values are `Club`, `Diamond`, `Heart`, and `Spade`, where `Ord(Club)` returns 0, `Ord(Diamond)` returns 1, and so forth.

When you declare an enumerated type, you are declaring each *val* to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock)
```

Unfortunately, `Click` is also the name of a method defined for `TControl` and all of the objects in VCL that descend from it. So if you're writing an application and you create an event handler like

```
procedure TForm1.DBGridEnter(Sender: TObject);
var Thing: TSound;
begin
  ...
  Thing := Click;
end;
```

you'll get a compilation error; the compiler interprets `Click` within the scope of the procedure as a reference to `TForm`'s `Click` method. You can work around this by qualifying the identifier; thus, if `TSound` is declared in `MyUnit`, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:


```

type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe)

```

You can use the (*val1*, ..., *valn*) construction directly in variable declarations, as if it were a type name:

```

var MyCard: (Club, Diamond, Heart, Spade);

```

But if you declare `MyCard` this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```

var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);

```

generates a compilation error. But

```

var Card1, Card2: (Club, Diamond, Heart, Spade);

```

compiles cleanly, as does

```

type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;

```

Enumerated Types with Explicitly Assigned Ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with = *constantExpression*, where *constantExpression* is a constant expression that evaluates to an integer. For example,

```

type Size = (Small = 5, Medium = 10, Large = Small + Medium);

```

defines a type called `Size` whose possible values include `Small`, `Medium`, and `Large`, where `Ord(Small)` returns 5, `Ord(Medium)` returns 10, and `Ord(Large)` returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration. In the previous example, the `Size` type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.)

Only three of these values have names, but the others are accessible through typecasts and through routines such as `Pred`, `Succ`, `Inc`, and `Dec`. In the following example, "anonymous" values in the range of `Size` are assigned to the variable `X`.

```
var X: Size;

X := Small;    // Ord(X) = 5
Y := Size(6);  // Ord(X) = 6
Inc(X);        // Ord(X) = 7
```

Any value that isn't explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn't assigned an ordinality, its ordinality is 0. Hence, given the declaration

```
type SomeEnum = (e1, e2, e3 = 1);
```

`SomeEnum` has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because `e2` and `e3` have the same ordinality, they represent the same value.

Enumerated constants without a specific value have RTTI:

```
type SomeEnum = (e1, e2, e3);
```

whereas enumerated constants with a specific value, such as the following, do not have RTTI:

```
type SomeEnum = (e1 = 1, e2 = 2, e3 = 3);
```

Subrange Types

A subrange type represents a subset of the values in another ordinal type (called the base type). Any construction of the form *Low..High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here `TMyColors` includes the values `Green`, `Yellow`, `Orange`, `Purple`, and `White`.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *LowerBound..UpperBound* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example, if `Color` is a variable that holds the value `Green`, `Ord(Color)` returns 2 regardless of whether `Color` is of type `TColors` or `TMyColors`.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type Percentile = 0..99;
var I: Percentile;
...
I := 100;
```

produces an error,

```
...
I := 99;
Inc(I);
```

assigns the value 100 to `I` (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after `=` is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;

type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real Types

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types on the Win32 platform.

Fundamental Win32 real types

Type	Range	Significant digits	Size in bytes
Real48	-2.9 x 10 ³⁹ .. 1.7 x 10 ³⁸	11-12	6
Single	-1.5 x 10 ⁴⁵ .. 3.4 x 10 ³⁸	7-8	4
Double	-5.0 x 10 ³²⁴ .. 1.7 x 10 ³⁰⁸	15-16	8
Extended	-3.6 x 10 ⁴⁹⁵¹ .. 1.1 x 10 ⁴⁹³²	10-20	10
Comp	-2 ⁶³⁺¹ .. 2 ⁶³ 1	10-20	8

Currency	-922337203685477.5808.. 922337203685477.5807	10-20	8
----------	--	-------	---

The following table shows how the fundamental real types map to .NET framework types.

Fundamental .NET real type mappings

Type	.NET Mapping
Real48	Deprecated
Single	Single
Double	Double
Extended	Double
Comp	Deprecated
Currency	Re-implemented as a value type using the Decimal type from the .NET Framework

The generic type Real, in its current implementation, is equivalent to Double (which maps to Double on .NET).

Generic real types

Type	Range	Significant digits	Size in bytes
Real	-5.0 x 10 ³²⁴ .. 1.7 x 10 ³⁰⁸	15–16	8

Note: The six-byte Real48 type was called Real in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte Real type in Delphi, you may want to change it to Real48. You can also use the `{ $REALCOMPATIBILITY ON }` compiler directive to turn Real back into the six-byte type.

The following remarks apply to fundamental real types.

- Real48 is maintained for backward compatibility. Since its storage format is not native to the Intel processor architecture, it results in slower performance than other floating-point types. The Real48 type has been deprecated on the .NET platform.
- Extended offers greater precision than other real types but is less portable. Be careful using Extended if you are creating data files to share across platforms.
- The Comp (computational) type is native to the Intel processor architecture and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a Comp value.) Comp is maintained for backward compatibility only. Use the Int64 type for better performance.
- Currency is a fixed-point data type that minimizes rounding errors in monetary calculations. On the Win32 platform, it is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, Currency values are automatically divided or multiplied by 10000.

String Types

This topic describes the string data types available in the Delphi language. The following types are covered:

- Short strings.
- Long strings.
- Wide (Unicode) strings.

About String Types

A string represents a sequence of characters. Delphi supports the following predefined string types.

String types

Type	Maximum length	Memory required	Used for
ShortString	255 characters	2 to 256 bytes	backward compatibility
AnsiString	$\sim 2^{31}$ characters	4 bytes to 2GB	8-bit (ANSI) characters, DBCS ANSI, MBCS ANSI, etc.
WideString	$\sim 2^{30}$ characters	4 bytes to 2GB	Unicode characters; multi-user servers and multi-language applications

On the Win32 platform, AnsiString, sometimes called the long string, is the preferred type for most purposes. WideString is the preferred string type on the .NET platform.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a function or procedure (as var and out parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type.

The reserved word string functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable `S` that holds a string. On the Win32 platform, the compiler interprets string (when it appears without a bracketed number after it) as AnsiString. On the .NET platform, the string type maps to the String class. You can use single byte character strings on the .NET platform, but you must explicitly declare them to be of type AnsiString.

On the Win32 platform, you can use the `{ $H- }` directive to turn string into ShortString. The `{ $H- }` directive is deprecated on the .NET platform.

The standard function `Length` returns the number of characters in a string. The `SetLength` procedure adjusts the length of a string.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value. For example, 'AB' is greater than 'A'; that is, 'AB' > 'A' returns True. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If `S` is a string variable and `i` an integer expression, `S[i]` represents the *i*th character - or, strictly speaking, the *i*th byte in `S`. For a ShortString or AnsiString, `S[i]` is of type AnsiChar; for a WideString, `S[i]` is of type WideChar. For single-byte (Western) locales, `MyString[2] := 'A'`; assigns the value `A` to the second character of `MyString`. The following code uses the standard `AnsiUpperCase` function to convert `MyString` to uppercase.

```
var I: Integer;
begin
  I := Length(MyString);
```

```

while I > 0 do
begin
  MyString[I] := AnsiUpperCase(MyString[I]);
  I := I - 1;
end;
end;

```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as var parameters, because this results in inefficient code.

You can assign the value of a string constant - or any other expression that returns a string - to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```

MyString := 'Hello world!';
MyString := 'Hello' + 'world';
MyString := MyString + '!';
MyString := ' '; { space }
MyString := ''; { empty string }

```

Short Strings

A ShortString is 0 to 255 characters long. While the length of a ShortString can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If *S* is a ShortString variable, `Ord(S[0])`, like `Length(S)`, returns the length of *S*; assigning a value to *S*[0], like calling `SetLength`, changes the length of *S*. ShortString is maintained for backward compatibility only.

The Delphi language supports short-string types - in effect, subtypes of ShortString - whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word string. For example,

```
var MyString: string[100];
```

creates a variable called `MyString` whose maximum length is 100 characters. This is equivalent to the declarations

```

type CString = string[100];
var MyString: CString;

```

Variables declared in this way allocate only as much memory as the type requires - that is, the specified maximum length plus one byte. In our example, `MyString` uses 101 bytes, as compared to 256 bytes for a variable of the predefined ShortString type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions `High` and `Low` operate on short-string type identifiers and variables. `High` returns the maximum length of the short-string type, while `Low` returns zero.

Long Strings

AnsiString, also called a long string, represents a dynamically allocated string whose maximum length is limited only by available memory.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty - that is, when it contains a zero-length string - the pointer is nil and the string uses no additional storage. When the variable is

nonempty, it points a dynamically allocated block of memory that contains the string value. The eight bytes before the location contain a 32-bit length indicator and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called reference-counting. When indexing is used to change the value of a single character in a string, a copy of the string is made if - but only if - its reference count is greater than one. This is called copy-on-write semantics.

WideString

The WideString type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to AnsiString. On Win32, WideString is compatible with the COM BSTR type.

Note: Under Win32, WideString values are not reference-counted.

The Win32 platform supports single-byte and multibyte character sets as well as Unicode. With a single-byte character set (SBCS), each byte in a string represents one character.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. The null value (#0) is always a single-byte character. Multibyte character sets - especially double-byte character sets (DBCS) - are widely used for Asian languages.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called wide characters and wide character strings. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2).

The Delphi language supports single-byte and multibyte characters and strings through the Char, PChar, AnsiChar, PAnsiChar, and AnsiString types. Indexing of multibyte strings is not reliable, since `s[i]` represents the *i*th byte (not necessarily the *i*th character) in `s`. However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with `Ansi-`. For example, the multibyte version of `StrPos` is `AnsiStrPos`.) Multibyte character support is operating-system dependent and based on the current locale.

Delphi supports Unicode characters and strings through the WideChar, PWideChar, and WideString types.

Working with null-Terminated Strings

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on null-terminated strings. A null-terminated string is a zero-based array of characters that ends with NUL (#0); since the array has no length indicator, the first NUL character marks the end of the string. You can use Delphi constructions and special routines in the `SysUtils` unit (see Standard routines and I/O) to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```

type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;

```

With extended syntax enabled (`{${X+}}`), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won't work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to `#0`.

Using Pointers, Arrays, and String Constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See [Pointers and pointer types](#).) String constants are assignment-compatible with the `PChar` and `PWideChar` types, which represent pointers to null-terminated arrays of `Char` and `WideChar` values. For example,

```

var P: PChar;
...
P := 'Hello world!'

```

points `P` to an area of memory that contains a null-terminated copy of 'Hello world!' This is equivalent to

```

const TempString: array[0..12] of Char = 'Hello world!';
var P: PChar;
...
P := @TempString[0];

```

You can also pass string constants to any function that takes value or `const` parameters of type `PChar` or `PWideChar` - for example `StrUpper('Hello world!')`. As with assignments to a `PChar`, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize `PChar` or `PWideChar` constants with string literals, alone or in a structured type. Examples:

```

const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = ('Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six',
    'Seven', 'Eight', 'Nine');

```

Zero-based character arrays are compatible with `PChar` and `PWideChar`. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,


```

var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;

```

This code calls `SomeProcedure` twice with the same value.

A character pointer can be indexed as if it were an array. In the previous example, `MyPointer[0]` returns `H`. The index specifies an offset added to the pointer before it is dereferenced. (For `PWideChar` variables, the index is automatically multiplied by two.) Thus, if `P` is a character pointer, `P[0]` is equivalent to `P^` and specifies the first character in the array, `P[1]` specifies the second character in the array, and so forth; `P[-1]` specifies the 'character' immediately to the left of `P[0]`. The compiler performs no range checking on these indexes.

The `StrUpper` function illustrates the use of pointer indexing to iterate through a null-terminated string:

```

function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
    begin
      Dest[I] := UpCase(Source[I]);
      Inc(I);
    end;
  Dest[I] := #0;
  Result := Dest;
end;

```

Mixing Delphi Strings and Null-Terminated Strings

You can mix long strings (AnsiString values) and null-terminated strings (PChar values) in expressions and assignments, and you can pass PChar values to functions or procedures that take long-string parameters. The assignment `S := P`, where `S` is a string variable and `P` is a PChar expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a PChar, the PChar operand is converted to a long string.

You can cast a PChar value as a long string. This is useful when you want to perform a string operation on two PChar values. For example,

```

S := string(P1) + string(P2);

```

You can also cast a long string as a null-terminated string. The following rules apply.

- If `S` is a long-string expression, `PChar(S)` casts `S` as a null-terminated string; it returns a pointer to the first character in `S`. For example, if `Str1` and `Str2` are long strings, you could call the Win32 API `MessageBox` function like this: `MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);`

- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if `S` is empty, the typecast returns `nil`.
- `PChar(S)` always returns a pointer to a memory block; if `S` is empty, a pointer to `#0` is returned.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the long string only when all of the following conditions are satisfied.
 - The expression cast is a long-string variable.
 - The string is not empty.
 - The string is unique - that is, has a reference count of one. To guarantee that the string is unique, call the `SetLength`, `SetString`, or `UniqueString` procedure.
 - The string has not been modified since the typecast was made.
 - The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing `WideString` values with `PWideChar` values.

Structured Types

Instances of a structured type hold more than one value. Structured types include sets, arrays, records, and files as well as class, class-reference, and interface types. Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

Note: Typed and untyped file types are not supported with the .NET framework.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word `packed` to implement compressed data storage. For example, `type TNumbers = packed array [1..100] of Real;`

Using `packed` slows data access and, in the case of a character array, affects type compatibility (for more information, see [Memory management](#)).

This topic covers the following structured types:

- Sets
- Arrays, including static and dynamic arrays.
- Records
- File types

Sets

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the base type; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

`set of baseType`

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

create a set type called `TIntSet` whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

```
type TIntSet = set of 1..250;
```

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
...
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

You can also use the `set of ...` construction directly in variable declarations:

```
var MySet: set of 'a'..'z';
...
MySet := ['a','b','c'];
```

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The `in` operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by `[]`.

Arrays

An array represents an indexed collection of elements of the same type (called the base type). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

Static Arrays

Static array types are denoted by constructions of the form

```
array[ indexType1, ..., indexTypeN ] of baseType;
```

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexTypes* index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexTypes*. In practice, *indexTypes* are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

```
var MyArray: array [1..100] of Char;
```

declares a variable called `MyArray` that holds an array of 100 character values. Given this declaration, `MyArray[3]` denotes the third character in `MyArray`. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type TMatrix = array[1..10, 1..50] of Real;
```

Whichever way `TMatrix` is declared, it represents an array of 500 real values. A variable `MyMatrix` of type `TMatrix` can be indexed like this: `MyMatrix[2,45]`; or like this: `MyMatrix[2][45]`. Similarly,

```
packed array[Boolean, 1..10, TShoeSize] of Integer;
```

is equivalent to

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

The standard functions `Low` and `High` operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function `Length` returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of `Char` values is called a packed string. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See [Type compatibility and identity](#).

An array type of the form `array[0..x] of Char` is called a zero-based character array. Zero-based character arrays are used to store null-terminated strings and are compatible with `PChar` values. See [Working with null-terminated strings](#).

Dynamic Arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the `SetLength` procedure. Dynamic-array types are denoted by constructions of the form

`array of baseType`

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for `MyFlexibleArray`. To create the array in memory, call `SetLength`. For example, given the previous declaration,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign `nil` to a variable that references the array or pass the variable to `Finalize`; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays are automatically released when their reference-count drops to zero. Dynamic arrays of length 0 have the value `nil`. Do not apply the dereference operator (`^`) to a dynamic-array variable or pass it to the `New` or `Dispose` procedure.

If `X` and `Y` are variables of the same dynamic-array type, `X := Y` points `X` to the same array as `Y`. (There is no need to allocate memory for `X` before performing this operation.) Unlike strings and static arrays, *copy-on-write* is not employed for dynamic arrays, so they are not automatically copied before they are written to. For example, after this code executes,

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
```

```
B[0] := 2;
end;
```

the value of `A[0]` is 2. (If `A` and `B` were static arrays, `A[0]` would still be 1.)

Assigning to a dynamic-array index (for example, `MyFlexibleArray[2] := 7`) does not reallocate the array. Out-of-range indexes are not reported at compile time.

In contrast, to make an independent copy of a dynamic array, you must use the global `Copy` function:

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := Copy(A);
  B[0] := 2; { B[0] <> A[0] }
end;
```

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

`A = B` returns False but `A[0] = B[0]` returns True.

To truncate a dynamic array, pass it to `SetLength`, or pass it to `Copy` and assign the result back to the array variable. (The `SetLength` procedure is usually faster.) For example, if `A` is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of `A`.

Once a dynamic array has been allocated, you can pass it to the standard functions `Length`, `High`, and `Low`. `Length` returns the number of elements in the array, `High` returns the array's highest index (that is, `Length - 1`), and `Low` returns 0. In the case of a zero-length array, `High` returns 1 (with the anomalous consequence that `High < Low`).

Note: In some function and procedure declarations, array parameters are represented as `array of baseType`, without any index types specified. For example, `function CheckStrings(A: array of string): Boolean;`

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically.

Multidimensional Dynamic Arrays

To declare multidimensional dynamic arrays, use iterated `array of ...` constructions. For example,

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

declares a two-dimensional array of strings. To instantiate this array, call `SetLength` with two integer arguments. For example, if `I` and `J` are integer-valued variables,

```
SetLength(Msgs, I, J);
```

allocates an *I*-by-*J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call `SetLength`, passing it parameters for the first *n* dimensions of the array. For example,

```
var Ints: array of array of Integer;
SetLength(Ints, 10);
```

allocates ten rows for `Ints` but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

```
SetLength(Ints[2], 5);
```

makes the third column of `Ints` five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column - for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the `IntToStr` function declared in the `SysUtils` unit) to create a triangular matrix of strings.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
    begin
      SetLength(A[I], I);
      for J := Low(A[I]) to High(A[I]) do
        A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
      end;
    end;
end;
```

Array Types and Assignments

Arrays are assignment-compatible only if they are of the same type. Because the Delphi language uses name-equivalence for types, the following code will not compile.

```
var
  Int1: array[1..10] of Integer;
  array[1..10] of Integer;
  ...
  Int1 := Int2;
```

To make the assignment work, declare the variables as

```
var Int1, Int2: array[1..10] of Integer;
```

or

```
type IntArray = array[1..10] of Integer;
var
  Int1: IntArray;
  Int2: IntArray;
```

Dynamically Allocated Multidimensional Arrays (.NET)

On the .NET platform, multidimensional arrays can be dynamically allocated using the New standard function. Using the New syntax to allocate an array, the array declaration specifies the number of dimensions, but not their actual size. You then pass the element type, the actual array dimensions, or an array initializer list to the New function. The array declaration has the following syntax:

```
array[, ..., ] of baseType;
```

In the syntax, the number of dimensions are specified by using a comma as a placeholder. The actual size is not determined until runtime when you call the New function. There are two forms of the New function: One takes the element type and the size of the array, and the other takes the element type and an array initializer list. The following code demonstrates both forms:

```
var
  a: array [, , ] of integer;  // 3 dimensional array
  b: array [, ] of integer;    // 2 dimensional array
  c: array [, ] of TPoint;     // 2 dimensional array of TPoint

begin
  a := New(array[3,5,7] of integer);           // New taking element type and size
of each dimension.
  b := New(array[,] of integer, ((1,2,3), (4,5,6))); // New taking the element type and
initializer list.
                                     // New taking an initializer list of TPoint.
  c := New(array[,] of TPoint, ((X:1;Y:2), (X:3;Y:4)), ((X:5;Y:6), (X:7;Y:8)));
end.
```

You can allocate the array by passing variable or constant expressions to the New function:

```
var
  a: array[,] of integer;
  r,c: Integer;

begin
  r := 4;
  c := 17;

  a := New(array [r,c] of integer);
```

You can also use the SetLength procedure to allocate the array by passing the array expression and the size of each dimension:


```

var
  a: array[,] of integer;
  b: array[,,] of integer;

begin
  SetLength(a, 4,5);
  SetLength(b, 3,5,7);
end.

```

The Copy function can be used to make a copy of an entire array. You cannot use Copy to duplicate a portion of an array.

You cannot pass a dynamically allocated rectangular array to the Low or High functions. This will generate a compile-time error.

Records (traditional)

A record (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called a field; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```

type recordTypeName = record
  fieldList1: type1;
  ...
  fieldListn: typen;
end

```

where *recordTypeName* is a valid identifier, each type denotes a type, and each fieldList is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called `TDateRec`.

```

type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;

```

Each `TDateRec` contains three fields: an integer value called `Year`, a value of an enumerated type called `Month`, and another integer between 1 and 31 called `Day`. The identifiers `Year`, `Month`, and `Day` are the field designators for `TDateRec`, and they behave like variables. The `TDateRec` type declaration, however, does not allocate any memory for the `Year`, `Month`, and `Day` fields; memory is allocated when you instantiate the record, like this:

```

var Record1, Record2: TDateRec;

```

This variable declaration creates two instances of `TDateRec`, called `Record1` and `Record2`.

You can access the fields of a record by qualifying the field designators with the record's name:

```

Record1.Year := 1904;

```

```
Record1.Month := Jun;  
Record1.Day := 16;
```

Or use a with statement:

```
with Record1 do  
begin  
    Year := 1904;  
    Month := Jun;  
    Day := 16;  
end;
```

You can now copy the values of `Record1`'s fields to `Record2`:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the `record ...` construction directly in variable declarations:

```
var S: record  
    Name: string;  
    Age: Integer;  
end;
```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

Variant Parts in Records

A record type can have a variant part, which looks like a case statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```
type recordTypeName = record  
    fieldList1: type1;  
    ...  
    fieldListn: typen;  
    case tag: ordinalType of  
        constantList1: (variant1);  
        ...  
        constantListn: (variantn);  
    end;
```

The first part of the declaration - up to the reserved word `case` - is the same as that of a standard record type. The remainder of the declaration - from `case` to the optional final semicolon - is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (:) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type *ordinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantLists*.

- Each *variant* is a semicolon-delimited list of declarations resembling the *fieldList: type* constructions in the main part of the record type. That is, a variant has the form

```
fieldList1: type1;  
...  
fieldListn: typen;
```

where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each type denotes a type, and the final semicolon is optional. The types must not be long strings, dynamic arrays, variants (that is, Variant types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several variants which share the same space in memory. You can read or write to any field of any variant at any time; but if you write to a field in one variant and then to a field in another variant, you may be overwriting your own data. The tag, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```
type  
  TEmployee = record  
    FirstName, LastName: string[40];  
    BirthDate: TDate;  
    case Salaried: Boolean of  
      True: (AnnualSalary: Currency);  
      False: (HourlyWage: Currency);  
  end;
```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of `TEmployee`, there is no reason to allocate enough memory for both fields. In this case, the only difference between the variants is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```

type
  TPerson = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
      True: (Birthplace: string[40]);
      False: (Country: string[20];
              EntryPort: string[20];
              EntryDate, ExitDate: TDate);
    end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
      Circle: (Radius: Real);
      Ellipse, Other: ();
    end;

```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest variant. The optional tag and the *constantLists* (like `Rectangle`, `Triangle`, and so forth in the last example) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit Real as the first field in one variant and a 32-bit Integer as the first field in another, you can assign a value to the Real field and then read back the first 32 bits of it as the value of the Integer field (passing it, say, to a function that requires integer parameters).

Records (advanced)

In addition to the traditional record types, the Delphi language allows more complex and “class-like” record types. In addition to fields, records may have properties and methods (including constructors), class properties, class methods, class fields, and nested types. For more information on these subjects, see the documentation on Classes and Objects. Here is a sample record type definition with some “class-like” functionality.

```

type
  TMyRecord = record
    type
      TInnerColorType = Integer;
    var
      Red: Integer;
    class var
      Blue: Integer;
    procedure printRed();
    constructor Create(val: Integer);
    property RedProperty: TInnerColorType read Red write Red;
    class property BlueProp: TInnerColorType read Blue write Blue;
  end;

constructor TMyRecord.Create(val: Integer);
begin
  Red := val;
end;

procedure TMyRecord.printRed;

```

```
begin
  writeln('Red: ', Red);
end;
```

Though records can now share much of the functionality of classes, there are some important differences between classes and records.

- Records do not support inheritance.
- Records can contain variant parts; classes cannot.
- Records are value types, so they are copied on assignment, passed by value, and allocated on the stack unless they are declared globally or explicitly allocated using the [New](#) and [Dispose](#) function. Classes are reference types, so they are not copied on assignment, they are passed by reference, and they are allocated on the heap.
- Records allow operator overloading on the Win32 and .NET platforms; classes allow operator overloading only for .NET.
- Records are constructed automatically, using a default no-argument constructor, but classes must be explicitly constructed. Because records have a default no-argument constructor, any user-defined record constructor must have one or more parameters.
- Record types cannot have destructors.
- Virtual methods (those specified with the `virtual`, `dynamic`, and `message` keywords) cannot be used in record types.
- Unlike classes, record types on the Win32 platform cannot implement interfaces; however, records on the .NET platform can implement interfaces.

File Types (Win32)

File types, which are available only on the Win32 platform, are sequences of elements of the same type. Standard I/O routines use the predefined [TextFile](#) or [Text](#) type, which represents a file containing characters organized into lines. For more information about file input and output, see [Standard routines and I/O](#).

To declare a file type, use the syntax

```
type fileTypeName =fileoftype
```

where *fileTypeName* is any valid identifier and *type* is a fixed-size type. Pointer types - whether implicit or explicit - are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```
type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;
  PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the `file of ...` construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word `file` by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see [Untyped files](#).

Files are not allowed in arrays or records.

Pointers and Pointer Types

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it points to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure. If that address is already taken, then the pointer holds the address to the first element.

Pointers are typed to indicate the kind of data stored at the addresses they hold. The general-purpose Pointer type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

This topic contains information on the following:

- General overview of pointer types.
- Declaring and using the pointer types supported by Delphi.

Overview of pointers

To see how pointers work, look at the following example.

```
1      var
2      X, Y: Integer; // X and Y are Integer variables
3      P: ^Integer    // P points to an Integer
4      begin
5          X := 17;     // assign a value to X
6          P := @X;     // assign the address of X to P
7          Y := P^;     // dereference P; assign the result to Y
8      end;
```

Line 2 declares `X` and `Y` as variables of type `Integer`. Line 3 declares `P` as a pointer to an `Integer` value; this means that `P` can point to the location of `X` or `Y`. Line 5 assigns a value to `X`, and line 6 assigns the address of `X` (denoted by `@X`) to `P`. Finally, line 7 retrieves the value at the location pointed to by `P` (denoted by `P^`) and assigns it to `Y`. After this code executes, `X` and `Y` have the same value, namely 17.

The `@` operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see [The @ operator and Procedural types in statements and expressions](#).

The symbol `^` has two purposes, both of which are illustrated in our example. When it appears before a type identifier

^typeName

it denotes a type that represents pointers to variables of type *typeName*. When it appears after a pointer variable

pointer^

it dereferences the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another - something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand the Delphi language, since pointers often operate behind the scenes in code where they don't appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class instance variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Delphi's strict data typing. By referencing a variable with an all-purpose `Pointer`, casting the `Pointer` to a more specific type, and then dereferencing it, you can treat the data

stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ...
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from `R` to `I`, without converting it.

In addition to assigning the result of an `@` operation, you can use several standard routines to give a value to a pointer. The `New` and `GetMem` procedures assign a memory address to an existing pointer, while the `Addr` and `Ptr` functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `P1^.Data^`.

The reserved word `nil` is a special constant that can be assigned to any pointer. When `nil` is assigned to a pointer, the pointer doesn't reference anything.

Pointer Types

You can declare a pointer to any type, using the syntax

```
type pointerTypeName = ^type
```

When you define a record or other data type, it might be useful to also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Note: You can declare a pointer type before you declare the type it points to.

Standard pointer types exist for many purposes. The most versatile is `Pointer`, which can point to data of any kind. But a `Pointer` variable cannot be dereferenced; placing the `^` symbol after a `Pointer` variable causes a compilation error. To access the data referenced by a `Pointer` variable, first cast it to another pointer type and then dereference it.

Character Pointers

The fundamental types `PAnsiChar` and `PWideChar` represent pointers to `AnsiChar` and `WideChar` values, respectively. The generic `PChar` represents a pointer to a `Char` (that is, in its current implementation, to an `AnsiChar`). These character pointers are used to manipulate null-terminated strings. (See [Working with null-terminated strings](#).)

Type-checked Pointers

The `$T` compiler directive controls the types of pointer values generated by the `@` operator. This directive takes the form of:


```
{ $T+ } or { $T- }
```

In the `{ $T- }` state, the result type of the `@` operator is always an untyped pointer that is compatible with all other pointer types. When `@` is applied to a variable reference in the `{ $T+ }` state, the type of the result is `^T`, where `T` is compatible only with pointers to the type of the variable.

Other Standard Pointer Types

The `System` and `SysUtils` units declare many standard pointer types that are commonly used.

Selected pointer types declared in `System` and `SysUtils`

Pointer type	Points to variables of type
PAnsiString, PString	AnsiString
PByteArray	TByteArray (declared in <code>SysUtils</code>). Used to typecast dynamically allocated memory for array access.
PCurrency, PDouble, PExtended, PSingle	Currency, Double, Extended, Single
PInteger	Integer
POleVariant	OleVariant
PShortString	ShortString. Useful when porting legacy code that uses the old PString type.
PTextBuf	TTextBuf (declared in <code>SysUtils</code>). TTextBuf is the internal buffer type in a TTextRec file record.)
PVarRec	TVarRec (declared in <code>System</code>)
PVariant	Variant
PWideString	WideString
PWordArray	TWordArray (declared in <code>SysUtils</code>). Used to typecast dynamically allocated memory for arrays of 2-byte values.

Procedural Types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions.

About Procedural Types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called `Calc` that takes two integer parameters and returns an integer:

```
function Calc(X,Y: Integer): Integer;
```

You can assign the `Calc` function to the variable `F`:

```
var F: function(X,Y: Integer): Integer;  
F := Calc;
```

If you take any procedure or function heading and remove the identifier after the word procedure or function, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the previous example) or to declare new types:

```
type  
  TIntegerFunction = function: Integer;  
  TProcedure = procedure;  
  TStrProc = procedure(const S: string);  
  TMathFunc = function(X: Double): Double;  
var  
  F: TIntegerFunction;    { F is a parameterless function that returns an integer }  
  Proc: TProcedure;       { Proc is a parameterless procedure }  
  SP: TStrProc;           { SP is a procedure that takes a string parameter }  
  M: TMathFunc;           { M is a function that takes a Double (real) parameter and returns  
a Double }  
  
  procedure FuncProc(P: TIntegerFunction); { FuncProc is a procedure whose only parameter  
is a parameterless integer-valued function }
```

On Win32, the variables shown in the previous example are all procedure pointers - that is, pointers to the address of a procedure or function. On the .NET platform, procedural types are implemented as delegates. If you want to reference a method of an instance object (see [Classes and objects](#)), you need to add the words `of object` to the procedural type name. For example

```
type  
  TMethod = procedure of object;  
  TNotifyEvent = procedure(Sender: TObject) of object;
```

These types represent method pointers. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

```
type  
  TNotifyEvent = procedure(Sender: TObject) of object;
```

```

TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ...
end;
var
    MainForm: TMainForm;
    OnClick: TNotifyEvent

```

we could make the following assignment.

```
OnClick := MainForm.ButtonClick;
```

Two procedural types are compatible if they have

- the same calling convention,
- the same return value (or no return value), and
- the same number of parameters, with identically typed parameters in corresponding positions. (Parameter names do not matter.)

On Win32, procedure pointer types are always incompatible with method pointer types, but this is not true on the .NET platform. The value nil can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like `Length` as a procedural value, write a wrapper for it:

```

function FLength(S: string): Integer;
begin
    Result := Length(S);
end;

```

Procedural Types in Statements and Expressions

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```

var
    F: function(X: Integer): Integer;
    I: Integer;
    function SomeFunction(X: Integer): Integer;
    ...
    F := SomeFunction;    // assign SomeFunction to F
    I := F(4);           // call function; assign result to I

```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```

var
    F, G: function: Integer;
    I: Integer;

```

```

function SomeFunction: Integer;
...
F := SomeFunction;      // assign SomeFunction to F
G := F;                 // copy F to G
I := G;                 // call function; assign result to I

```

The first statement assigns a procedural value to `F`. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to `I`. Because `I` is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of `F` results in a function call; the compiler calls the function pointed to by `F`, then calls the function `MyFunction`, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where `F` references a procedure (which doesn't return a value), or where `F` references a function that requires parameters, the previous statement causes a compilation error. To compare the procedural value of `F` with `MyFunction`, use

```
if @F = @MyFunction then ...;
```

`@F` converts `F` into an untyped pointer variable that contains an address, and `@MyFunction` returns the address of `MyFunction`.

To get the memory address of a procedural variable (rather than the address stored in it), use `@@`. For example, `@@F` returns the address of `F`.

The `@` operator can also be used to assign an untyped pointer value to a procedural variable. For example,

```

var StrComp: function(Str1, Str2: PChar): Integer;
...
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');

```

calls the `GetProcAddress` function and points `StrComp` to the result.

Any procedural variable can hold the value `nil`, which means that it points to nothing. But attempting to call a nil-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function `Assigned`:

```
if Assigned(OnClick) then OnClick(X);
```


Variant Types

This topic discusses the use of variant data types.

Variants Overview

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type `Variant`, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See [Object interfaces](#).) They can hold dynamic arrays, and they can hold a special kind of static array called a variant array. (See [Variant arrays](#).) Variants can mix with other variants and with integer, real, string, and Boolean values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if `v` is a variant that holds a string value, the construction `v[1]` causes a runtime error.

You can define custom Variants that extend the `Variant` type to hold arbitrary values. For example, you can define a Variant string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the `TCustomVariantType` class.

Note: This, and almost all variant functionality, is implemented in the `Variants` unit.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value `Unassigned`. The special value `Null` indicates unknown or missing data.

The standard function `VarType` returns a variant's type code. The `varTypeMask` constant is a bit mask used to extract the code from `VarType`'s return value, so that, for example,

```
VarType(V) and varTypeMask = varDouble
```

returns `True` if `v` contains a `Double` or an array of `Double`. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The `TVarData` record type defined in the `System` unit can be used to typecast variants and gain access to their internal representation.

Variant Type Conversions

All integer, real, string, character, and Boolean types are assignment-compatible with `Variant`. Expressions can be explicitly cast as variants, and the `VarAsType` and `VarCast` standard routines can be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
```

```

begin
  V1 := 1;           { integer value }
  V2 := 1234.5678; { real value }
  V3 := 'Hello world!'; { string value }
  V4 := '1000';       { string value }
  V5 := V1 + V2 + V4; { real value 2235.5678}
  I  := V1;           { I = 1 (integer value) }
  D  := V2;           { D = 1234.5678 (real value) }
  S  := V3;           { S = 'Hello world!' (string value) }
  I  := V4;           { I = 1000 (integer value) }
  S  := V5;           { S = '2235.5678' (string value) }
end;

```

The compiler performs type conversions according to the following rules.

Variant type conversion rules

Target	integer	real	string	Boolean
Source				
integer	converts integer formats	converts to real	converts to string representation	returns False if 0, True otherwise
real	rounds to nearest integer	converts real formats	converts to string representation using regional settings	returns False if 0, True otherwise
string	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/character formats	returns False if string is 'false' (noncase-sensitive) or a numeric string that evaluates to 0, True if string is 'true' or a nonzero numeric string; raises exception otherwise
character	same as string (above)	same as string (above)	same as string (above)	same as string (above)
Boolean	False = 0, True: all bits set to 1 (-1 if Integer, 255 if Byte, etc.)	False = 0, True = 1	False = 'False', True = 'True' by default; casing depends on global variable BooleanToStringRule	False = False, True = True
Unassigned	returns 0	returns 0	returns empty string	returns False
Null	depends on global variable NullStrictConvert (raises an exception by default)	depends on global variable NullStrictConvert (raises an exception by default)	depends on global variables NullStrictConvert and NullAsStringValue (raises an exception by default)	depends on global variable NullStrictConvert (raises an exception by default)

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid variant operations, assignments or casts raise an `EVariantError` exception or an exception class descending from `EVariantError`.

Special conversion rules apply to the `TDateTime` type declared in the `System` unit. When a `TDateTime` is converted to any other type, it is treated as a normal `Double`. When an integer, real, or Boolean is converted to a `TDateTime`, it is first converted to a `Double`, then read as a date-time value. When a string is converted to a `TDateTime`, it is interpreted as a date-time value using the regional settings. When an `Unassigned` value is converted to `TDateTime`, it is treated like the real or integer value 0. Converting a `Null` value to `TDateTime` raises an exception.

On the Win32 platform, if a variant references a COM interface, any attempt to convert it reads the object's default property and converts that value to the requested type. If the object has no default property, an exception is raised.

Variants in Expressions

All operators except `^`, `is`, and `in` take variant operands. Except for comparisons, which always return a Boolean result, any operation on a variant value returns a variant result. If an expression combines variants with statically-typed values, the statically-typed values are automatically converted to variants.

This is not true for comparisons, where any operation on a Null variant produces a `Null` variant. For example:

```
V := Null + 3;
```

assigns a `Null` variant to `V`. By default, comparisons treat the `Null` variant as a unique value that is less than any other value. For example:

```
if Null > -3 then ... else ...;
```

In this example, the `else` part of the `if` statement will be executed. This behavior can be changed by setting the `NullEqualityRule` and `NullMagnitudeRule` global variables.

Variant Arrays

You cannot assign an ordinary static array to a variant. Instead, create a variant array by calling either of the standard functions `VarArrayCreate` or `VarArrayOf`. For example,

```
V: Variant;  
...  
V := VarArrayCreate([0,9], varInteger);
```

creates a variant array of integers (of length 10) and assigns it to the variant `V`. The array can be indexed using `V[0]`, `V[1]`, and so forth, but it is not possible to pass a variant array element as a `var` parameter. Variant arrays are always indexed with integers.

The second parameter in the call to `VarArrayCreate` is the type code for the array's base type. For a list of these codes, see `VarType`. Never pass the code `varString` to `VarArrayCreate`; to create a variant array of strings, use `varOleStr`.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except `ShortString` and `AnsiString`, and if the base type of the array is `Variant`, its elements can even be heterogeneous. Use the `VarArrayRedim` function to resize a variant array. Other standard routines that operate on variant arrays include `VarArrayDimCount`, `VarArrayLowBound`, `VarArrayHighBound`, `VarArrayRef`, `VarArrayLock`, and `VarArrayUnlock`.

Note: Variant arrays of custom variants are not supported, as instances of custom variants can be added to a `VarVariant` variant array.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

OleVariant

The `OleVariant` type exists on both the Windows and Linux platforms. The main difference between *Variant* and `OleVariant` is that *Variant* can contain data types that only the current application knows what to do with. `OleVariant` can only contain the data types defined as compatible with OLE Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Delphi string, or a one of the new custom variant types) to an OleVariant, the runtime library tries to convert the *Variant* into one of the OleVariant standard data types (such as a Delphi string converts to an OLE BSTR string). For example, if a variant containing an AnsiString is assigned to an OleVariant, the AnsiString becomes a WideString. The same is true when passing a *Variant* to an OleVariant function parameter.

Type Compatibility and Identity

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include:

- Type identity
- Type compatibility
- Assignment compatibility

Type Identity

When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

`T1`, `T2`, `T3`, `T4`, and `Integer` all denote the same type. To create distinct types, repeat the word `type` in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called `TMyInteger` which is not identical to `Integer`.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, `TS1` and `TS2`. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```
type MyString = string[10];
var
```

```
S1: MyString;  
S2: MyString;
```

Type Compatibility

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of characters.
- One is a string type and the other is a string, packed-string, or Char type.
- One type is Variant and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is PChar or PWideChar and the other is a zero-based character array of the form `array[0..n]` of PChar or PWideChar.
- One type is Pointer (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the `{ $T+ }` compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

Assignment Compatibility

Assignment-compatibility is not a symmetric relation. An expression of type T2 can be assigned to a variable of type T1 if the value of the expression falls in the range of T1 and at least one of the following conditions is satisfied.

- T1 and T2 are of the same type, and it is not a file type or structured type that contains a file type at any level.
- T1 and T2 are compatible ordinal types.
- T1 and T2 are both real types.
- T1 is a real type and T2 is an integer type.
- T1 is PChar, PWideChar or any string type and the expression is a string constant.
- T1 and T2 are both string types.
- T1 is a string type and T2 is a Char or packed-string type.
- T1 is a long string and T2 is PChar or PWideChar.
- T1 and T2 are compatible packed-string types.
- T1 and T2 are compatible set types.
- T1 and T2 are compatible pointer types.
- T1 and T2 are both class, class-reference, or interface types and T2 is a derived from T1.
- T1 is an interface type and T2 is a class type that implements T1.
- T1 is PChar or PWideChar and T2 is a zero-based character array of the form `array[0..n]` of Char (when T1 is PChar) or of WideChar (when T1 is PWideChar).

- T1 and T2 are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type.)
- T1 is Variant and T2 is an integer, real, string, character, Boolean, interface type or OleVariant type.
- T1 is an OleVariant and T2 is an integer, real, string, character, Boolean, interface, or Variant type.
- T1 is an integer, real, string, character, or Boolean type and T2 is Variant or OleVariant.
- T1 is the `IUnknown` or `IDispatch` interface type and T2 is Variant or OleVariant. (The variant's type code must be `varEmpty`, `varUnknown`, or `varDispatch` if T1 is `IUnknown`, and `varEmpty` or `varDispatch` if T1 is `IDispatch`.)

Declaring Types

This topic describes the syntax of Delphi type declarations.

Type Declaration Syntax

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

`type newTypeName = type`

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier's scope doesn't include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

X and *Y* are of the same type; at runtime, there is no way to distinguish *TValue* from *Real*. This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type information for example, to associate a property editor with properties of a particular type - the distinction between 'different name' and 'different type' becomes important. In this case, use the syntax

`type newTypeName = type type`

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called *TValue*.

For *var* parameters, types of formal and actual must be identical. For example,

```
type
  TMyType = type Integer
  procedure p(var t: TMyType);
  begin
  end;

procedure x;
var
  m: TMyType;
  i: Integer;
begin
```

```
p(m); // Works  
p(i); // Error! Types of formal and actual must be identical.  
end;
```

Note: This only applies to `var` parameters, not to `const` or by-value parameters.

Variables

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

Declaring Variables

The basic syntax for a variable declaration is

```
var identifierList : type;
```

where *identifierList* is a comma-delimited list of valid identifiers and *type* is any valid type. For example,

```
var I: Integer;
```

declares a variable `I` of type `Integer`, while

```
var X, Y: Real;
```

declares two variables - `X` and `Y` - of type `Real`.

Consecutive variable declarations do not have to repeat the reserved word `var`:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called local, while other variables are called global. Global variables can be initialized at the same time they are declared, using the syntax

```
var identifier: type = constantExpression;
```

where *constantExpression* is any constant expression representing a value of type *type*. Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
...
I := 7;
```

Local variables cannot be initialized in their declarations. Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Object instance data (fields) are also initialized to 0. On the Win32 platform, the contents of a local variable are undefined until a value is assigned to them. On the .NET platform, the CLR initializes all variables, including local variables, to 0.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see [Memory management](#).

Absolute Addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive `absolute` after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

specifies that the variable `StrLen` should start at the same address as `Str`. Since the first byte of a short string contains the string's length, the value of `StrLen` is the length of `Str`.

You cannot initialize a variable in an absolute declaration or combine `absolute` with any other directives.

Dynamic Variables

You can create dynamic variables by calling the `GetMem` or `New` procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use `FreeMem` to destroy variables created by `GetMem` and `Dispose` to destroy variables created by `New`. Other standard routines that operate on dynamic variables include `ReallocMem`, `AllocMem`, `Initialize`, `Finalize`, `StrAlloc`, and `StrDispose`.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

Thread-local Variables

Thread-local (or thread) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with `threadvar` instead of `var`. For example,

```
threadvar X: Integer;
```

Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the `absolute` directive.

Dynamic variables that are ordinarily managed by the compiler (long strings, wide strings, dynamic arrays, variants, and interfaces) can be declared with `threadvar`, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory from within the thread, before the thread terminates. For example,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
...
S := ''; // free the memory used by S
```

Note: Use of such constructs is discouraged.

You can free a variant by setting it to `Unassigned` and an interface or dynamic array by setting it to `nil`.

Declared Constants

Several different language constructions are referred to as 'constants'. There are numeric constants (also called numerals) like 17, and string constants (also called character strings or string literals) like 'Hello world!'. Every enumerated type defines constants that represent the values of that type. There are predefined constants like True, False, and nil. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

True Constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called `MaxValue` that returns the integer 237. The syntax for declaring a true constant is

`const identifier = constantExpression`

where `identifier` is any valid identifier and `constantExpression` is an expression that the compiler can evaluate without executing your program.

If `constantExpression` returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called `MyNumber`, of type `Int64`, that returns the integer 17. Otherwise, the type of the declared constant is the type of the `constantExpression`.

- If `constantExpression` is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.
- If `constantExpression` is a real, its type is `Extended`. If it is an integer, its type is given by the table below.

Types for integer constants

Range of constant(hexadecimal)	Range of constant(decimal)	Type
-\$8000000000000000..-\$80000001	-2 ⁶³ ..-2147483649	Int64
-\$80000000..-\$8001	-2147483648..-32769	Integer
-\$8000..-\$81	-32768..-129	Smallint
-\$80..-1	-128..-1	Shortint
0..\$7F	0..127	0..127
\$80..\$FF	128..255	Byte
\$0100..\$7FFF	256..32767	0..32767
\$8000..\$FFFF	32768..65535	Word
\$10000..\$7FFFFFFF	65536..2147483647	0..2147483647
\$80000000..\$FFFFFFFF	2147483648..4294967295	Cardinal
\$100000000..\$FFFFFFFFFFFFFFFF	4294967296..2 ⁶³ -1	Int64

Here are some examples of constant declarations:

```

const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;

```

Constant Expressions

A constant expression is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants True, False, and nil; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

Abs	High	Low	Pred	Succ
Chr	Length	Odd	Round	Swap
Hi	Lo	Ord	SizeOf	Trunc

This definition of a constant expression is used in several places in Delphi's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing case statements, and declaring both true and typed constants.

Examples of constant expressions:

```

100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1

```

Resource Strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program.

Resource strings are declared like other true constants, except that the word `const` is replaced by `resourcestring`. The expression to the right of the `=` symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';
  OpenError = 'Cannot open file %s';
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks';
  SomeResourceString = SomeTrueConstant;
```

Typed Constants

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

Declare a typed constant like this:

`const identifier: type = value`

where *identifier* is any valid identifier, *type* is any type except files and variants, and *value* is an expression of type *type*. For example,

```
const Max: Integer = 100;
```

In most cases, *value* must be a constant expression; but if *type* is an array, record, procedural, or pointer type, special rules apply.

Array Constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called `Digits` that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the previous declaration can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

creates an array called `Maze` where

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
```

```
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

Array constants cannot contain file-type values at any level.

Record Constants

To declare a record constant, specify the value of each field - as `fieldName: value`, with the field assignments separated by semicolons - in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record constants cannot contain file-type values at any level.

Procedural Constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```
function Calc(X, Y: Integer): Integer;
begin
  ...
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Given these declarations, you can use the procedural constant `MyFunction` in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value `nil` to a procedural constant.

Pointer Constants

When you declare a pointer constant, you must initialize it to a value that can be resolved at least as a relative address at compile time. There are three ways to do this: with the `@` operator, with `nil`, and (if the constant is of type

PChar or PWideChar) with a string literal. For example, if `I` is a global variable of type Integer, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a PChar constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```


Procedures and Functions

This section describes the syntax of function and procedure declarations.

In This Section

[Procedures and Functions](#)

Describes the basics of procedure and function declarations in Delphi.

[Parameters](#)

Describes the usage of function and procedure parameter lists.

[Calling Procedures and Functions](#)

Describes program control, the definition of var and out parameters, open array constructors, and the inline directive.

Procedures and Functions

This topic covers the following items:

- Declaring procedures and functions
- Calling conventions
- Forward and interface declarations
- Declaration of external routines
- Overloading procedures and functions
- Local declarations and nested routines

About Procedures and Functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls `SomeFunction` and assigns the result to `I`. Function calls cannot appear on the left side of an assignment statement.

Procedure calls - and, when extended syntax is enabled (`{ $X+ }`), function calls - can be used as complete statements. For example,

```
DoSomething;
```

calls the `DoSomething` routine; if `DoSomething` is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

Declaring Procedures and Functions

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the prototype, heading, or header. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's body or block.

Procedure Declarations

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;  
  localDeclarations;  
begin  
  statements  
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

Here is an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Given this declaration, you can call the `NumString` procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value '17' to `MyString` (which must be a string variable).

Within a procedure's statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like `N` and `S` in the previous example); the parameter list defines a set of local variables, so don't try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

Function Declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```
function functionName(parameterList): returnType; directives;
  localDeclarations;
begin
  statements
end;
```

where *functionName* is any valid identifier, *returnType* is a type identifier, *statements* is a sequence of statements that execute when the function is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable `Result`.

As long as extended syntax is enabled (`{ $X+ }`), `Result` is implicitly declared in every function. Do not try to redeclare it.

For example,

```
function WF: Integer;
begin
```

```
WF := 17;
end;
```

defines a constant function called `WF` that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
    Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
    X: Real;
    I: Integer;
begin
    X := A[0];
    for I := 1 to N - 1 do
        if X < A[I] then X := A[I];
    end;
    Max := X;
end;
```

You can assign a value to `Result` or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to `Result` or to the function name becomes the function's return value. For example,

```
function Power(X: Real; Y: Integer): Real;
var
    I: Integer;
begin
    Result := 1.0;
    I := Y;
    while I > 0 do
        begin
            if Odd(I) then Result := Result * X;
            I := I div 2;
            X := Sqr(X);
        end;
    end;
end;
```

`Result` and the function name always represent the same value. Hence

```
function MyFunction: Integer;
begin
    MyFunction := 5;
    Result := Result * 2;
    MyFunction := Result + 1;
end;
```

returns the value 11. But `Result` is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like `Result`) to track

the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. Result, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

If the function exits without assigning a value to Result or the function name, then the function's return value is undefined.

Calling Conventions

When you declare a procedure or function, you can specify a calling convention using one of the directives `register`, `pascal`, `cdecl`, `stdcall`, and `safecall`. For example,

```
function MyFunction(X, Y: Real): Real; cdecl;
```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is `register`.

- The `register` and `pascal` conventions pass parameters from left to right; that is, the left most parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The `cdecl`, `stdcall`, and `safecall` conventions pass parameters from right to left.
- For all conventions except `cdecl`, the procedure or function removes parameters from the stack upon returning. With the `cdecl` convention, the caller removes parameters from the stack when the call returns.
- The `register` convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The `safecall` convention implements exception 'firewalls.' On Win32, this implements interprocess COM error notification.

The table below summarizes calling conventions.

Calling conventions

Directive	Parameter order	Clean-up	Passes parameters in registers?
register	Left-to-right	Routine	Yes
pascal	Left-to-right	Routine	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Routine	No
safecall	Right-to-left	Routine	No

The default `register` convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use `register`.) The `cdecl` convention is useful when you call functions from shared libraries written in C or C++, while `stdcall` and `safecall` are recommended, in general, for calls to external code. On Win32, the operating system APIs are `stdcall` and `safecall`. Other operating systems generally use `cdecl`. (Note that `stdcall` is more efficient than `cdecl`.)

The `safecall` convention must be used for declaring dual-interface methods. The `pascal` convention is maintained for backward compatibility.

The directives `near`, `far`, and `export` refer to calling conventions in 16-bit Windows programming. They have no effect in Win32, or in .NET applications and are maintained for backward compatibility only.

Forward and Interface Declarations

The forward directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called `Calculate`. Somewhere after the forward declaration, the routine must be redeclared in a defining declaration that includes a block. The defining declaration for `Calculate` might look like this:

```
function Calculate;  
... { declarations }  
begin  
... { statement block }  
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the forward declaration exactly (except that default parameters can be omitted). If the forward declaration specifies an overloaded procedure or function, then the defining declaration must repeat the parameter list.

A forward declaration and its defining declaration must appear in the same type declaration section. That is, you can't add a new section (such as a var section or const section) between the forward declaration and the defining declaration. The defining declaration can be an external or assembler declaration, but it cannot be another forward declaration.

The purpose of a forward declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the forward-declared routine before it is actually defined. Besides letting you organize your code more flexibly, forward declarations are sometimes necessary for mutual recursions.

The forward directive has no effect in the interface section of a unit. Procedure and function headers in the interface section behave like forward declarations and must have defining declarations in the implementation section. A routine declared in the interface section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

External Declarations

The external directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C function that takes a variable number of parameters, use the `varargs` directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The `varargs` directive works only with external routines and only with the `cdecl` calling convention.

Linking to Object Files

To call routines from a separately compiled object file, first link the object file to your application using the `$L` (or `$LINK`) compiler directive. For example,

```
{ $L BLOCK.OBJ }
```

links BLOCK.OBJ into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;  
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the `MoveWord` and `FillWord` routines from BLOCK.OBJ.

On the Win32 platform, declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Delphi source code.

Importing Functions from Libraries

To import routines from a dynamically loadable library (.DLL), attach a directive of the form

```
external stringConstant;
```

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Win32

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called `SomeFunction` from `strlib.dll`.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the external directive:

```
external stringConstant1namestringConstant2;
```

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine's original name.

The following declaration imports a function from `user32.dll` (part of the Win32 API).

```
function MessageBox(HWND: Integer; Text, Caption: PChar; Flags: Integer): Integer; stdcall;  
external 'user32.dll' name 'MessageBoxA';
```

The function's original name is `MessageBoxA`, but it is imported as `MessageBox`.

Instead of a name, you can use a number to identify the routine you want to import:

```
externalstringConstantindexintegerConstant;
```

where *integerConstant* is the routine's index in the export table.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

Overloading Procedures and Functions

You can declare more than one routine in the same scope with the same name. This is called overloading. Overloaded routines must be declared with the overload directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;  
begin  
    Result := X/Y;
```



```

end

function Divide(X, Y: Integer): Integer; overload;
begin
    Result := X div Y;
end;

```

These declarations create two functions, both called `Divide`, that take parameters of different types. When you call `Divide`, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first `Divide` function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types - for example,

```

procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;

```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type `Extended`.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```

function Cap(S: string): string; overload;
...
procedure Cap(var Str: string); overload;
...

```

But the declarations

```

function Func(X: Real; Y: Integer): Real; overload;
...
function Func(X: Integer; Y: Real): Real; overload;
...

```

are legal.

When an overloaded routine is declared in a forward or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain `AnsiString/PChar` and `WideString/WideChar` parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is `AnsiString/PChar`.

```

procedure test(const S: String); overload;
procedure test(const W: WideString); overload;
var
    a: string;
    b: widestring;
begin
    a := 'a';
    b := 'b';

```

```

test(a);    // calls String version
test(b);    // calls WideString version
test('abc'); // calls String version
test(WideString('abc')); // calls widestring version
end;

```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```

procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);    // integer version
  foo(v);    // variant version
  foo(1.2);  // variant version (float literals -> extended precision)
end;

```

This example calls the variant version of `foo`, not the double version, because the 1.2 constant is implicitly an extended type and extended is not an exact match for double. Extended is also not an exact match for Variant, but Variant is considered a more general type (whereas double is a smaller type than extended).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```

const d: double = 1.2;
begin
  foo(d);
end;

```

The above code works correctly, and calls the double version.

```

const s: single = 1.2;
begin
  foo(s);
end;

```

The above code also calls the double version of `foo`. Single is a better fit to double than to variant.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (Single, Double, Extended) along with the variant version.

If you use default parameters in overloaded routines, be careful not to introduce ambiguous parameter signatures.

You can limit the potential effects of overloading by qualifying a routine's name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in `Unit1`; if no routine in `Unit1` matches the name and parameter list in the call, an error results.

Local Declarations

The body of a function or procedure often begins with declarations of local variables used in the routine's statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

Nested Routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called `DoSomething` contains a nested procedure.

```
procedure DoSomething(S: string);
var
    X, Y: Integer;

procedure NestedProc(S: string);
begin
    ...
end;

begin
    ...
    NestedProc(S);
    ...
end;
```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, `NestedProc` can be called only within `DoSomething`.

For real examples of nested routines, look at the `DateTimeToString` procedure, the `ScanDate` function, and other routines in the `SysUtils` unit.

Parameters

This topic covers the following items:

- Parameter semantics
- String parameters
- Array parameters
- Default parameters

About Parameters

Most procedure and function headers include a parameter list. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is `(X: Real; Y: Integer)`.

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the `=` symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by `var`, `const`, or `out`. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWND: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
  ...
end;
```

Within the procedure or function body, the parameter names (`X` and `Y` in the first example) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

Parameter Semantics

Parameters are categorized in several ways:

- Every parameter is classified as value, variable, constant, or out. Value parameters are the default; the reserved words `var`, `const`, and `out` indicate variable, constant, and out parameters, respectively.
- Value parameters are always typed, while constant, variable, and out parameters can be either typed or untyped.
- Special rules apply to array parameters.

Files and instances of structured types that contain files can be passed only as variable (`var`) parameters.

Value and Variable Parameters

Most parameters are either value parameters (the default) or variable (var) parameters. Value parameters are passed by value, while variable parameters are passed by reference. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;    // X is a value parameter
begin
    X := X * 2;
    Result := X;
end;

function DoubleByRef(var X: Integer): Integer; // X is a variable parameter
begin
    X := X * 2;
    Result := X;
end;
```

These functions return the same result, but only the second one - `DoubleByRef` can change the value of a variable passed to it. Suppose we call the functions like this:

```
var
    I, J, V, W: Integer;
begin
    I := 4;
    V := 4;
    J := DoubleByValue(I);    // J = 8, I = 4
    W := DoubleByRef(V);      // W = 8, V = 8
end;
```

After this code executes, the variable `I`, which was passed to `DoubleByValue`, has the same value we initially assigned to it. But the variable `V`, which was passed to `DoubleByRef`, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more var parameters, no copies are made. This is illustrated in the following example.

```
procedure AddOne(var X, Y: Integer);
begin
    X := X + 1;
    Y := Y + 1;
end;

var I: Integer;
begin
    I := 1;
    AddOne(I, I);
end;
```

After this code executes, the value of `i` is 3.

If a routine's declaration specifies a var parameter, you must pass an assignable expression - that is, a variable, typed constant (in the `{ $J+ }` state), dereferenced pointer, field, or indexed variable to the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in var parameters - for example, `DoubleByRef(MyArray[i])` - are evaluated once, before execution of the routine.

Constant Parameters

A constant (const) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a var parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using const allows the compiler to optimize code for structured - and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the `CompareStr` function in the `SysUtils` unit:

```
function CompareStr(const S1, S2: string): Integer;
```

Because `S1` and `S2` are not modified in the body of `CompareStr`, they can be declared as constant parameters.

Out Parameters

An out parameter, like a variable parameter, is passed by reference. With an out parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The out parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```
procedure GetInfo(out Info: SomeRecordType);
```

When you call `GetInfo`, you must pass it a variable of type `SomeRecordType`:

```
var MyRecord: SomeRecordType;  
...  
GetInfo(MyRecord);
```

But you're not using `MyRecord` to pass any data to the `GetInfo` procedure; `MyRecord` is just a container where you want `GetInfo` to store the information it generates. The call to `GetInfo` immediately frees the memory used by `MyRecord`, before program control passes to the procedure.

Out parameters are frequently used with distributed-object models like COM and CORBA. In addition, you should use out parameters when you pass an uninitialized variable to a function or procedure.

Untyped Parameters

You can omit type specifications when declaring var, const, and out parameters. (Value parameters must be typed.) For example,

```
procedure TakeAnything(const C);
```

declares a procedure called `TakeAnything` that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called `Equal` that compares a specified number of bytes of any two variables.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N : Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

Given the declarations

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

you could make the following calls to `Equal`:

```
Equal(Vec1, Vec2, SizeOf(TVector));           // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N);       // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5); // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4);                          // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

String Parameters

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```
procedure Check(S: string[20]); // syntax error
```

causes a compilation error. But

```
type TString20 = string[20];
procedure Check(S: TString20);
```


is valid. The special identifier `OpenString` can be used to declare routines that take short-string parameters of varying length:

```
procedure Check(S: OpenString);
```

When the `{H}` and `{P+}` compiler directives are both in effect, the reserved word `string` is equivalent to `OpenString` in parameter declarations.

Short strings, `OpenString`, `$H`, and `$P` are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

Array Parameters

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer) // syntax error<
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. Another approach is to use open array parameters.

Since the Delphi language does not implement value semantics for dynamic arrays, 'value' parameters in routines do not represent a full copy of the dynamic array. In this example

```
type
  TDynamicArray = array of Integer;
  procedure p(Value: TDynamicArray);
  begin
    Value[0] := 1;
  end;

  procedure Run;
  var
    a: TDynamicArray;
  begin
    SetLength(a, 1);
    a[0] := 0;
    p(a);
    Writeln(a[0]); // Prints '1'
  end;
```

Note that the assignment to `Value[0]` in routine `p` will modify the content of dynamic array of the caller, despite `Value` being a by-value parameter. If a full copy of the dynamic array is required, use the `Copy` standard procedure to create a value copy of the dynamic array.

Open Array Parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of type` (rather than `array[X..Y] of type`) in the parameter declaration. For example,

```
function Find(A: array of Char): Integer;
```

declares a function called `Find` that takes a character array of any size and returns an integer.

Note: The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The previous example creates a function that takes any array of Char elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;  
function Find(A: TDynamicCharArray): Integer;
```

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard `Low` and `High` functions return 0 and `Length1`, respectively. The `SizeOf` function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped var parameters. They cannot be passed to `SetLength`.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a `Clear` procedure that assigns zero to each element in an array of reals and a `Sum` function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);  
var  
    I: Integer;  
begin  
    for I := 0 to High(A) do A[I] := 0;  
end;  
  
function Sum(const A: array of Real): Real;  
var  
    I: Integer;  
    S: Real;  
begin  
    S := 0;  
    for I := 0 to High(A) do S := S + A[I];  
    Sum := S;  
end;
```

When you call routines that use open array parameters, you can pass open array constructors to them.

Variant Open Array Parameters

Variant open array parameters allow you to pass an array of differently typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify `array of const` as the parameter's type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called `DoSomething` that can operate on heterogeneous arrays.

On the .NET platform, a variant open array parameter is equivalent to `array of TObject`. To determine the type of an element in the array, you may use the `ClassName` or `GetType` methods.

On the Win32 platform, the `array of const` construction is equivalent to `array of TVarRec`. `TVarRec`, declared in the `System` unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. `TVarRec`'s `VType` field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as `Pointer` and must be typecast to string.

The following Win32 example, uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in `SysUtils`. This function will not compile on .NET because it depends on the variant implementation of `array of const`.

```
function MakeStr(const Args: array of const): string;
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:  Result := Result + IntToStr(VInteger);
        vtBoolean:  Result := Result + BoolToStr(VBoolean);
        vtChar:     Result := Result + VChar;
        vtExtended: Result := Result + FloatToStr(VExtended^);
        vtString:   Result := Result + VString^;
        vtPChar:    Result := Result + VPChar;
        vtObject:   Result := Result + VObject.ClassName;
        vtClass:    Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:  Result := Result + CurrToStr(VCurrency^);
        vtVariant:  Result := Result + string(VVariant^);
        vtInt64:    Result := Result + IntToStr(VInt64^);
      end;
    end;
  end;
```

We can call this function using an open array constructor. For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string `'test100 T3.14159TForm'`.

Default Parameters

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed const and value parameters. To provide a default value, end the parameter declaration with the `=` symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);  
FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real; // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string); // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;  
function Resizer(X: Real; Y: Real = 2.0): Real;  
var  
  F: TResizer;  
  N: Real;
```

the statements

```
F := Resizer;  
F(N);
```

result in the values (N, 1.0) being passed to `Resizer`.

Default parameters are limited to values that can be specified by a constant expression. Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than nil as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

Default Parameters and Overloaded Functions

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;  
...  
procedure Confused(I: Integer; J: Integer = 0); overload;  
...  
Confused(X);    // Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

Default Parameters in Forward and Interface Declarations

If a routine has a forward declaration or appears in the interface section of a unit, default parameter values if there are any must be specified in the forward or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the forward or interface declaration exactly.

Calling Procedures and Functions

This topic covers the following items:

- Program control and routine parameters
- Open array constructors
- The inline directive

Program Control and Parameters

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called actual parameters, while the parameters in the routine's declaration are called formal parameters.

When calling a routine, remember that

- expressions used to pass typed const and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass var and out parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass var and out parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(,,X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();  
DoSomething;
```

Open Array Constructors

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.

For example, given the declarations

```
var I, J: Integer;  
procedure Add(A: array of Integer);
```

you could call the [Add](#) procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;  
...  
Temp[0] := 5;  
Temp[1] := 7;  
Temp[2] := I;  
Temp[3] := I + J;  
Add(Temp);
```

Open array constructors can be passed only as value or const parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

Using the inline Directive

The Delphi compiler allows functions and procedures to be tagged with the inline directive to improve performance. If the function or procedure meets certain criteria, the compiler will insert code directly, rather than generating a call. Inlining is a performance optimization that can result in faster code, but at the expense of space. Inlining always causes the compiler to produce a larger binary file. The inline directive is used in function and procedure declarations and definitions, like other directives.

```
procedure MyProc(x:Integer); inline;  
begin  
    // ...  
end;  
  
function MyFunc(y:Char) : String; inline;  
begin  
    // ..  
end;
```

The inline directive is a suggestion to the compiler. There is no guarantee the compiler will inline a particular routine, as there are a number of circumstances where inlining cannot be done. The following list shows the conditions under which inlining does or does not occur:

- Inlining will not occur on any form of late-bound method. This includes virtual, dynamic, and message methods.
- Routines containing assembly code will not be inlined.
- Constructors and destructors will not be inlined.
- The main program block, unit initialization, and unit finalization blocks cannot be inlined.
- Routines that are not defined before use cannot be inlined.
- Routines that take open array parameters cannot be inlined.
- Code can be inlined within packages, however, inlining never occurs across package boundaries.
- No inlining will be done between units that are circularly dependent. This included indirect circular dependencies, for example, unit A uses unit B, and unit B uses unit C which in turn uses unit A. In this example, when compiling unit A, no code from unit B or unit C will be inlined in unit A.

- The compiler can inline code when a unit is in a circular dependency, as long as the code to be inlined comes from a unit outside the circular relationship. In the above example, if unit A also used unit D, code from unit D could be inlined in A, since it is not involved in the circular dependency.
- If a routine is defined in the interface section and it accesses symbols defined in the implementation section, that routine cannot be inlined.
- In Delphi.NET, routines in classes cannot be inlined if they access members with less (i.e. more restricted) visibility than the method itself. For example, if a public method accesses private symbols, it cannot be inlined.
- If a routine marked with inline uses external symbols from other units, all of those units must be listed in the uses statement, otherwise the routine cannot be inlined.
- Procedures and functions used in conditional expressions in while-do and repeat-until statements cannot be expanded inline.
- Within a unit, the body for an inline function should be defined before calls to the function are made. Otherwise, the body of the function, which is not known to the compiler when it reaches the call site, cannot be expanded inline.

If you modify the implementation of an inlined routine, you will cause all units that use that function to be recompiled. This is different from traditional rebuild rules, where rebuilds were triggered only by changes in the interface section of a unit.

The `{ $INLINE }` compiler directive gives you finer control over inlining. The `{ $INLINE }` directive can be used at the site of the inlined routine's definition, as well as at the call site. Below are the possible values and their meaning:

Value	Meaning at definition	Meaning at call site
<code>{ \$INLINE ON }</code> (default)	The routine is compiled as inlineable if it is tagged with the inline directive.	The routine will be expanded inline if possible.
<code>{ \$INLINE AUTO }</code>	Behaves like <code>{ \$INLINE ON }</code> , with the addition that routines not marked with inline will be inlined if their code size is less than or equal to 32 bytes.	<code>{ \$INLINE AUTO }</code> has no effect on whether a routine will be inlined when it is used at the call site of the routine.
<code>{ \$INLINE OFF }</code>	The routine will not be marked as inlineable, even if it is tagged with inline.	The routine will not be expanded inline.

Classes and Objects

This section describes the object-oriented features of the Delphi language, such as the declaration and usage of class types.

In This Section

[Classes and Objects](#)

Presents a conceptual overview of classes and class types in the Delphi language.

[Fields](#)

Describes the syntax of class data field declarations.

[Methods](#)

Describes the syntax of procedure and function declarations within class types.

[Properties](#)

Describes the syntax of property declarations in Delphi classes.

[Events](#)

Describes how to use events in Delphi classes.

[Class References](#)

Describes operations and methods on class types, as opposed to instances (i.e. objects).

[Exceptions](#)

Describes the usage of exceptions and exception handlers.

[Nested Type Declarations](#)

Describes the syntax of declaring and using nested types within Delphi class declarations.

[Operator Overloading](#)

Describes the syntax of operator overloading in Delphi.

[Class Helpers](#)

Describes the syntax of class helpers.

Classes and Objects

This topic covers the following material:

- Declaration syntax of classes
- Inheritance and scope
- Visibility of class members
- Forward declarations and mutually dependent classes

Class Types

A class, or class type, defines a structure consisting of fields, methods, and properties. Instances of a class type are called objects. The fields, methods, and properties of a class are called its components or members.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class' fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects that is, instances of a class. Some methods (called class methods) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have access specifiers, which determine how their data is read and modified. From other parts of a program outside of the object itself a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called constructors and destructors.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value `nil`. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the `Size` property of the object referenced by `SomeObject`; you would not write this as `SomeObject^.Size := 100`.

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type
  className = class [abstract | sealed] (ancestorClass)
    memberList
  end;
```

where *className* is any valid identifier, the `sealed` or `abstract` keyword is optional, (*ancestorClass*) is optional, and *memberList* declares members - that is, fields, methods, and properties - of the class. If you omit (*ancestorClass*), then the new class inherits directly from the predefined `TObject` class. If you include (*ancestorClass*) and *memberList* is empty, you can omit `end`. A class type declaration can also include a list of interfaces implemented by the class; see [Implementing Interfaces](#).

If a class is marked `sealed`, then it cannot be extended through inheritance. If a class is marked `abstract`, then it cannot be instantiated directly using the `Create` constructor. An entire class can be declared `abstract` even if it does not contain any abstract virtual methods. A class cannot be both `abstract` and `sealed`.

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the TMemoryStream class from the [Classes](#) unit.

```
type TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
end;
```

TMemoryStream descends from TCustomMemoryStream (in the [Classes](#) unit), inheriting most of its members. But it defines - or redefines - several methods and properties, including its destructor method, [Destroy](#). Its constructor, [Create](#), is inherited without change from TObject, and so is not redeclared. Each member is declared as private, protected, or public (this class has no published members). These terms are explained below.

Given this declaration, you can create an instance of TMemoryStream as follows:

```
var stream: TMemoryStream;
    stream := TMemoryStream.Create;
```

Inheritance and Scope

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called [TSomeControl](#) that descends from TControl. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence [TSomeControl](#) contains all of the members defined in TControl and in each of TControl's ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

TObject and TClass

The TObject class, declared in the [System](#) unit, is the ultimate ancestor of all other classes. TObject defines only a handful of methods, including a basic constructor and destructor. In addition to TObject, the [System](#) unit declares the class reference type TClass:

```
TClass = class of TObject;
```

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from `TObject`. Thus

```
type TMyClass = class
    ...
end;
```

is equivalent to

```
type TMyClass = class(TObject)
    ...
end;
```

The latter form is recommended for readability.

Compatibility of Class Types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
    TFigure = class(TObject);
    TRectangle = class(TFigure);
    TSquare = class(TRectangle);
var
    Fig: TFigure;
```

the variable `Fig` can be assigned values of type `TFigure`, `TRectangle`, and `TSquare`.

Object Types

The Win32 Delphi compiler allows an alternative syntax to class types, which you can declare object types using the syntax

```
type objectType = object (ancestorObjectType)
    memberList
end;
```

where *objectTypeName* is any valid identifier, (*ancestorObjectType*) is optional, and *memberList* declares fields, methods, and properties. If (*ancestorObjectType*) is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from `TObject`, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the `New` procedure and destroy them with the `Dispose` procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended on Win32, and they have been completely deprecated in the Delphi for .NET compiler.

Visibility of Class Members

Every member of a class has an attribute called visibility, which is indicated by one of the reserved words `private`, `protected`, `public`, `published`, or `automated`. For example,

```
published property Color: TColor read GetColor write SetColor;
```

declares a published property called `Color`. Visibility determines where and how a member can be accessed, with `private` representing the least accessibility, `protected` representing an intermediate level of accessibility, and `public`, `published`, and `automated` representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default `published`, provided the class is compiled in the `{ $M+ }` state or is derived from a class compiled in the `{ $M+ }` state; otherwise, such members are `public`.

For readability, it is best to organize a class declaration by visibility, placing all the `private` members together, followed by all the `protected` members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new 'section' of the declaration. So a typical class declaration should like this:

```
type
  TMyClass = class(TControl)
    private
      ... { private declarations here }
    protected
      ... { protected declarations here }
    public
      ... { public declarations here }
    published
      ... { published declarations here }
  end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a `protected` property can be made `public` in a descendant, but not `private`. Moreover, `published` members cannot become `public` in a descendant class. For more information, see [Property overrides and redeclarations](#).

Private, Protected, and Public Members

A `private` member is invisible outside of the unit or program where its class is declared. In other words, a `private` method cannot be called from another module, and a `private` field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another's `private` members without making those members more widely accessible.

A `protected` member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. A `protected` method can be called, and a `protected` field or property read or written to, from the definition of any method belonging to a class that descends from the one where the `protected` member is declared. Members that are intended for use only in the implementation of derived classes are usually `protected`.

A `public` member is visible wherever its class can be referenced.

Strict Visibility Specifiers

In addition to `private` and `protected` visibility specifiers, the Delphi compiler supports additional visibility settings with greater access constraints. These settings are `strict private` and `strict protected` visibility. These settings strictly comply with the .NET Common Language Specification (CLS), and they can also be used in Win32 applications.

Class members with strict private visibility are accessible only within the class in which they are declared. They are not visible to procedures or functions declared within the same unit. Class members with strict protected visibility are visible within the class in which they are declared, and within any descendant class, regardless of where it is declared. Furthermore, when instance members (those declared without the `class` or `class var` keywords) are declared strict private or strict protected, they are inaccessible outside of the instance of a class in which they appear. An instance of a class cannot access strict protected or strict protected instance members in other instances of the same class.

Delphi's traditional private visibility specifier maps to the CLR's `assembly` visibility. Delphi's protected visibility specifier maps to the CLR's `assembly` or `family` visibility.

Note: The word *strict* is treated as a directive within the context of a class declaration. Within a class declaration you cannot declare a member named 'strict', but it is acceptable for use outside of a class declaration.

Published Members

Published members have the same visibility as public members. The difference is that runtime type information (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the **Object Inspector**, and to associate specific methods (called event handlers) with specific properties (called events).

Published properties are restricted to certain data types. Ordinal, string, class, interface, variant, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except Real48 can be published. Properties of an array type (as distinct from array properties, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties of all publishable types, and properties of enumerated types that include anonymous values. If you publish a property of this kind, the **Object Inspector** won't display it correctly, nor will the property's value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the `{ $M+ }` state or descends from a class compiled in the `{ $M+ }` state. Most classes with published members derive from `TPersistent`, which is compiled in the `{ $M+ }` state, so it is seldom necessary to use the `$M` directive.

Note: Identifiers that contain Unicode characters are not allowed in published sections of classes, or in types used by published members.

Automated Members (Win32 Only)

Automated members have the same visibility as public members. The difference is that Automation type information (required for Automation servers) is generated for automated members. Automated members typically appear only in Win32 classes, and the automated reserved word has been deprecated in the .NET compiler. The automated reserved word is maintained for backward compatibility. The `TAutoObject` class in the `ComObj` unit does not use automated.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool, and all interface types.
- Method declarations must use the default register calling convention. They can be virtual, but not dynamic.

- Property declarations can include access specifiers (read and write) but other specifiers (index, stored, default, and nodefault) are not allowed. Access specifiers must list a method identifier that uses the default register calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a `dispid` directive. Specifying an already used ID in a `dispid` directive causes an error.

On the Win32 platform, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Win32 only), see Automation objects.

Forward Declarations and Mutually Dependent Classes

If the declaration of a class type ends with the word `class` and a semicolon - that is, if it has the form

```
type className = class;
```

with no ancestor or class members listed after the word `class`, then it is a forward declaration. A forward declaration must be resolved by a defining declaration of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```
type
    TFigure = class;    // forward declaration
    TDrawing = class
        Figure: TFigure;
        ...
    end;

    TFigure = class    // defining declaration
        Drawing: TDrawing;
        ...
    end;
```

Do not confuse forward declarations with complete declarations of types that derive from `TObject` without declaring any class members.

```
type
    TFirstClass = class;    // this is a forward declaration
    TSecondClass = class    // this is a complete class declaration
    end;
    TThirdClass = class(TObject); // this is a complete class declaration
```

Fields

This topic describes the syntax of class data fields declarations.

About Fields

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. For example, the following declaration creates a class called `TNumber` whose only member, other than the methods it inherits from `TObject`, is an integer field called `Int`.

```
type
  TNumber = class
    var
      Int: Integer;
  end;
```

The `var` keyword is optional. However, if it is not used, then all field declarations must occur before any property or method declarations. After any property or method declarations, the `var` may be used to introduce any additional field declarations.

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```
type
  TAncestor = class
    Value: Integer;
  end;

  TDescendant = class(TAncestor)
    Value: string;    // hides the inherited Value field
  end;

var
  MyObject: TAncestor;

begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!'    // error

  (MyObject as TDescendant).Value := 'Hello!'    // works!
end;
```

Although `MyObject` holds an instance of `TDescendant`, it is declared as `TAncestor`. The compiler therefore interprets `MyObject.Value` as referring to the (integer) field declared in `TAncestor`. Both fields, however, exist in the `TDescendant` object; the inherited `Value` is hidden by the new one, and can be accessed through a typecast.

Constants, and typed constant declarations can appear in classes and non-anonymous records at global scope. Both constants and typed constants can also appear within nested type definitions. Constants and typed constants can appear only within class definitions when the class is defined locally to a procedure (i.e. they cannot appear within records defined within a procedure).

Class Fields

Class fields are data fields in a class that can be accessed without an object reference (unlike the normal “instance fields” which are discussed above). The data stored in a class field are shared by all instances of the class and may be accessed by referring to the class or to a variable that represents an instance of the class.

You can introduce a block of class fields within a class declaration by using the class var block declaration. All fields declared after class var have static storage attributes. A class var block is terminated by the following:

- 1 Another class var or var declaration
- 2 A procedure or function (i.e. method) declaration (including class procedures and class functions)
- 3 A property declaration (including class properties)
- 4 A constructor or destructor declaration
- 5 A visibility scope specifier (public, private, protected, published, strict private, and strict protected)

For example:

```
type
  TMyClass = class
    public
      class var          // Introduce a block of class static fields.
        Red: Integer;
        Green: Integer;
        Blue: Integer;
      var                // Ends the class var block.
        InstanceField: Integer;
  end;
```

The class fields `Red`, `Green`, and `Blue` can be accessed with the code:

```
TMyClass.Red := 1;
TMyClass.Green := 2;
TMyClass.Blue := 3;
```

Class fields may also be accessed through an instance of the class. With the following declaration:

```
var
  myObject: TMyClass;
```

This code has the same effect as the assignments to `Red`, `Green`, and `Blue` above:

```
myObject.Red := 1;
myObject.Green := 2;
myObject.Blue := 3;
```

Methods

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example, `SomeObject.Free` calls the `Free` method in `SomeObject`.

This topic covers the following material:

- Methods declarations and implementation
- Method binding
- Overloading methods
- Constructors and destructors
- Message methods

About Methods

Within a class declaration, methods appear as procedure and function headings, which work like forward declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of `TMyClass` includes a method called `DoSomething`:

```
type
  TMyClass = class(TObject)
    ...
    procedure DoSomething;
    ...
  end;
```

A defining declaration for `DoSomething` must occur later in the module:

```
procedure TMyClass.DoSomething;
begin
  ...
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class' methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does, the order, type and names of the parameters must match exactly, and if the method is a function, the return value must match as well.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

`reintroduce`; `overload`; *binding*; *calling convention*; `abstract`; *warning*

where *binding* is `virtual`, `dynamic`, or `override`; *calling convention* is `register`, `pascal`, `cdecl`, `stdcall`, or `safecall`; and *warning* is `platform`, `deprecated`, or `library`.

Inherited

The reserved word `inherited` plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If `inherited` is followed by the name of a member, it represents a normal method call or reference to a property or field - except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited `Create`.

When `inherited` has no identifier after it, it refers to the inherited method with the same name as the enclosing method or, if the enclosing method is a message handler, to the inherited message handler for the same message. In this case, `inherited` takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

Self

Within the implementation of a method, the identifier `Self` references the object in which the method is called. For example, here is the implementation of `TCollection`'s `Add` method in the `Classes` unit.

```
function TCollection.Add: TCollectionItem;  
begin  
    Result := FItemClass.Create(Self);  
end;
```

The `Add` method calls the `Create` method in the class referenced by the `FItemClass` field, which is always a `TCollectionItem` descendant. `TCollectionItem.Create` takes a single parameter of type `TCollection`, so `Add` passes it the `TCollection` instance object where `Add` is called. This is illustrated in the following code.

```
var MyCollection: TCollection;  
...  
MyCollection.Add    // MyCollection is passed to the TCollectionItem.Create method
```

`Self` is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class' methods. In this case, you can access the original member identifier as `Self.Identifier`.

For information about `Self` in class methods, see [Class methods](#).

Method Binding

Method bindings can be static (the default), virtual, or dynamic. Virtual and dynamic methods can be overridden, and they can be abstract. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

Static Methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the `Draw` methods are static.

```
type
  TFigure = class
    procedure Draw;
  end;

  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to `Figure.Draw`, the `Figure` variable references an object of class `TRectangle`, but the call invokes the implementation of `Draw` in `TFigure`, because the declared type of the `Figure` variable is `TFigure`.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;

begin
  Figure := TFigure.Create;
  Figure.Draw;           // calls TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;           // calls TFigure.Draw

  TRectangle(Figure).Draw; // calls TRectangle.Draw

  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;         // calls TRectangle.Draw
  Rectangle.Destroy;
end;
```

Virtual and Dynamic Methods

To make a method virtual or dynamic, include the virtual or dynamic directive in its declaration. Virtual and dynamic methods, unlike static methods, can be overridden in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method call—not the declared type of the variable—determines which implementation to activate.

To override a method, redeclare it with the override directive. An override declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the `Draw` method declared in `TFigure` is overridden in two descendant classes.

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
```

```

TRectangle = class(TFigure)
    procedure Draw; override;
end;

TEllipse = class(TFigure)
    procedure Draw; override;
end;

```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```

var
    Figure: TFigure;

begin
    Figure := TRectangle.Create;
    Figure.Draw;           // calls TRectangle.Draw
    Figure.Destroy;
    Figure := TEllipse.Create;
    Figure.Draw;           // calls TEllipse.Draw
    Figure.Destroy;
end;

```

Only virtual and dynamic methods can be overridden. All methods, however, can be overloaded; see [Overloading methods](#).

The Delphi compiler also supports the concept of a *final* virtual method. When the keyword *final* is applied to a virtual method, no descendent class can override that method. Use of the *final* keyword is an important design decision that can help document how the class is intended to be used. It can also give the compiler hints that allow it to optimize the code it produces.

Virtual Versus Dynamic

In Delphi for .NET, virtual and dynamic methods are identical. In Delphi for Win32, virtual and dynamic methods are semantically equivalent. However, they differ in the implementation of method-call dispatching at runtime: virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

Note: Only use dynamic methods if there is a clear, observable benefit. Generally, use virtual methods.

Overriding Versus Hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn't include *override*, the new declaration merely hides the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```

type
    T1 = class(TObject)
        procedure Act; virtual;
    end;

    T2 = class(T1)

```



```

    procedure Act;    // Act is redeclared, but not overridden
end;

var
    SomeObject: T1;

begin
    SomeObject := T2.Create;
    SomeObject.Act;    // calls T1.Act
end;

```

Reintroduce

The reintroduce directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```

procedure DoSomething; reintroduce;    // the ancestor class also has a DoSomething method

```

Use reintroduce when you want to hide an inherited virtual method with a new one.

Abstract Methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive `abstract` after `virtual` or `dynamic`. For example,

```

procedure DoSomething; virtual; abstract;

```

You can call an abstract method only in a class or instance of a class in which the method has been overridden.

Note: The Delphi for .NET compiler allows an entire class to be declared abstract, even though it does not contain any virtual abstract methods. See [Class Types](#) for more information.

Class Methods

Most methods are called instance methods, because they operate on an individual instance of an object. A class method is a method (other than a constructor) that operates on classes instead of objects. There are two types of class methods: ordinary class methods and class static methods.

Ordinary Class Methods

The definition of a class method must begin with the reserved word `class`. For example,

```

type
    TFigure = class
    public
        class function Supports(Operation: string): Boolean; virtual;
        class procedure GetInfo(var Info: TFigureInfo); virtual;
        ...
    end;

```

The defining declaration of a class method must also begin with `class`. For example,

```

class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
    ...
end;

```

In the defining declaration of a class method, the identifier `Self` represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class `C`, then `Self` is of the type class of `C`. Thus you cannot use the `Self` to access instance fields, instance properties, and normal (object) methods, but you can use it to call constructors and other class methods, or to access class properties and class fields.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of `Self`.

Class Static Methods

Like class methods, class static methods can be accessed without an object reference. Unlike ordinary class methods, class static methods have no `Self` parameter at all. They also cannot access any instance members. (They still have access to class fields, class properties, and class methods.) Also unlike class methods, class static methods cannot be declared virtual.

Methods are made class static by appending the word `static` to their declaration, for example

```

type
    TMyClass = class
    strict private
        class var
            FX: Integer;

    strict protected

        // Note: accessors for class properties must be declared class static.
        class function GetX: Integer; static;
        class procedure SetX(val: Integer); static;

    public
        class property X: Integer read GetX write SetX;
        class procedure StatProc(s: String); static;
end;

```

Like a class method, you can call a class static method through the class type (i.e. without having an object reference), for example

```

TMyClass.X := 17;
TMyClass.StatProc('Hello');

```

Overloading Methods

A method can be redeclared using the overload directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the reintroduce directive when you redeclare it in descendant classes. For example,

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;

  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
  ...

SomeObject := T2.Create;
SomeObject.Test('Hello!');           // calls T2.Test
SomeObject.Test(7);                  // calls T1.Test
```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type
  TSomeClass = class
    published
      function Func(P: Integer): Integer;
      function Func(P: Boolean): Integer; // error
    ...
```

Methods that serve as property read or write specifiers cannot be overloaded.

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see [Overloading procedures and functions](#).

Constructors

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word `constructor`. Examples:

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Constructors must use the default register calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor `Create`.

To create an object, call the constructor method on a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object, sets the values of all ordinal fields to zero, assigns nil to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the [Destroy](#) destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor's implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word `inherited` to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ...
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);      // Initialize inherited parts
  Width := 65;                 // Change inherited properties
  Height := 65;
  FPen := TPen.Create; // Initialize new fields
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), nil (pointer and class types), empty (string types), or Unassigned (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as virtual is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects that is, construction of objects whose types aren't known at compile time. (See [Class references](#).)

Note: For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic [Memory Management Issues on the .NET Platform](#).

The Class Constructor (.NET)

A class constructor executes before a class is referenced or used. The class constructor must be declared as strict private, and there can be at most one class constructor declared in a class. Descendants can declare their own class constructor, however, do not call inherited within the body of a class constructor. In fact, you cannot call a class constructor directly, or access it in any way (such as taking its address). The compiler generates code to call class constructors for you.

There can be no guarantees on when a class constructor will execute, except to say that it will execute at some time before the class is used. On the .NET platform in order for a class to be "used", it must reside in code that is actually

executed. For example, if a class is first referenced in an if statement, and the test of the if statement is never true during the course of execution, then the class will never be loaded and JIT compiled. Hence, in this case the class constructor would not be called.

The following class declaration demonstrates the syntax of class properties and fields, as well as class static methods and class constructors:

```
type
  TMyClass = class
    strict protected

    // Accessors for class properties must be declared class static.
    class function GetX: Integer; static;
    class procedure SetX(val: Integer); static;
  public
    class property X: Integer read GetX write SetX;
    class procedure StatProc(s: String); static;
  strict private
    class var
      FX: Integer;
    class constructor Create;
end;
```

Destructors

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word `destructor`. Example:

```
destructor SpecialDestructor(SaveData: Boolean);
destructor Destroy; override;
```

Destructors on Win32 must use the default register calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited `Destroy` method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, `Destroy` is automatically called to dispose of the unfinished object. This means that `Destroy` must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always nil. A destructor should therefore check for nil values before operating on class-type or pointer-type fields. Calling the `Free` method (defined in `TObject`), rather than `Destroy`, offers a convenient way of checking for nil values before destroying an object.

Note: For more information on constructors, destructors, and memory management issues on the .NET platform, please see the topic [Memory Management Issues on the .NET Platform](#).

Message Methods

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. VCL uses message methods to respond to Windows messages.

A message method is created by including the message directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Win32 message IDs defined, along with corresponding record types, in the [Messages](#) unit. A message method must be a procedure that takes a single `var` parameter.

For example:

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ...
  end;
```

A message method does not have to include the `override` directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

Implementing Message Methods

The implementation of a message method can call the inherited message method, as in this example:

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Message.CharCode = Ord(#13) then
    ProcessEnter
  else
    inherited;
end;
```

The inherited statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, inherited calls the `DefaultHandler` method originally defined in `TObject`.

The implementation of `DefaultHandler` in `TObject` simply returns without performing any actions. By overriding `DefaultHandler`, a class can implement its own default handling of messages. On Win32, the `DefaultHandler` method for controls calls the Win32 API `DefWindowProc`.

Message Dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the `Dispatch` method inherited from `TObject`:

```
procedure Dispatch(var Message);
```

The `Message` parameter passed to `Dispatch` must be a record whose first entry is a field of type `Word` containing a message ID.

`Dispatch` searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, `Dispatch` calls `DefaultHandler`.

Properties

This topic describes the following material:

- Property access
- Array properties
- Index specifiers
- Storage specifiers
- Property overrides and redeclarations
- Class properties

About Properties

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

```
property propertyName[indexes]: type index integerConstant specifiers;
```

where

- *propertyName* is any valid identifier.
- [*indexes*] is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier1*, ..., *identifiern*: type. For more information, see Array Properties, below.
- type must be a predefined or previously declared type identifier. That is, property declarations like `property Num: 0..9 ...` are invalid.
- the index *integerConstant* clause is optional. For more information, see Index Specifiers, below.
- *specifiers* is a sequence of read, write, stored, default (or no default), and implements specifiers. Every property declaration must have at least one read or write specifier.

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as var parameters, nor can the @ operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a read method that retrieves a value from a database or generates a random value.

Property Access

Every property has a read specifier, a write specifier, or both. These are called access specifiers and they have the form

read *fieldOrMethod*

write *fieldOrMethod*

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.

- If *fieldOrMethod* is a method, it cannot be dynamic and, if virtual, cannot be overloaded. Moreover, access methods for a published property must use the default register calling convention.
- In a read specifier, if *fieldOrMethod* is a method, it must be a parameterless function whose result type is the same as the property's type. (An exception is the access method for an indexed property or an array property.)
- In a write specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or const parameter of the same type as the property (or more, if it is an array property or indexed property).

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the `GetColor` method must be declared as

```
function GetColor: TColor;
```

and the `SetColor` method must be declared as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(The name of `SetColor`'s parameter, of course, doesn't have to be `Value`.)

When a property is referenced in an expression, its value is read using the field or method listed in the read specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the write specifier.

The example below declares a class called `TCompass` with a published property called `Heading`. The value of `Heading` is read through the `FHeading` field and written through the `SetHeading` procedure.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    ...
  end;
```

Given this declaration, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the `TCompass` class, no action is associated with reading the `Heading` property; the read operation consists of retrieving the value stored in the `FHeading` field. On the other hand, assigning a value to the `Heading` property translates into a call to the `SetHeading` method, which, presumably, stores the new value in the `FHeading` field as well as performing other actions. For example, `SetHeading` might be implemented like this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
    if FHeading <> Value then
    begin
        FHeading := Value;
        Repaint;    // update user interface to reflect new value
    end;
end;
```

A property whose declaration includes only a read specifier is a read-only property, and one whose declaration includes only a write specifier is a write-only property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

Array Properties

Array properties are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a read specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a write specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or const parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by indexing the property identifier. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
```

```
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\BIN');
```

The definition of an array property can be followed by the default directive, in which case the array property becomes the default property of the class. For example,

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ...
  end;
```

If a class has a default property, you can access that property with the abbreviation `object[index]`, which is equivalent to `object.property[index]`. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property with a given signature (array parameter list), but it is possible to overload the default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always binds to properties statically.

Index Specifiers

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive `index` followed by an integer constant between -2147483647 and 2147483647. If a property has an index specifier, its read and write specifiers must list methods rather than fields. For example,

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
    property Left: Longint index 0 read GetCoordinate write SetCoordinate;
    property Top: Longint index 1 read GetCoordinate write SetCoordinate;
    property Right: Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
    ...
  end;
```

An access method for a property with an index specifier must take an extra value parameter of type `Integer`. For a read function, it must be the last parameter; for a write procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if `Rectangle` is of type `TRectangle`, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage Specifiers

The optional stored, default, and nodefault directives are called storage specifiers. They have no effect on program behavior, but control whether or not to save the values of published properties in form files.

The stored directive must be followed by True, False, the name of a Boolean field, or the name of a parameterless method that returns a Boolean value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no stored directive, it is treated as if stored True were specified.

The default directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited default value without specifying a new one, use the nodefault directive. The default and nodefault directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without default or nodefault, it is treated as if nodefault were specified. For reals, pointers, and strings, there is an implicit default value of 0, nil, and '' (the empty string), respectively.

Note: You can't use the ordinal value 2147483648 as a default value. This value is used internally to represent nodefault.

When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its default value (or if there is no default value) and the stored specifier is True, then the property's value is saved. Otherwise, the property's value is not saved.

Note: Property values are not automatically initialized to the default value. That is, the default directive controls only when property values are saved to the form file, but not the initial value of the property on a newly created instance.

Storage specifiers are not supported for array properties. The default directive has a different meaning when used in an array property declaration. See Array Properties, above.

Property Overrides and Redeclarations

A property declaration that doesn't specify a type is called a property override. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word property followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as protected, a derived class can redeclare it in a public or published section of the class. Property overrides can include read, write, stored, default, and nodefault directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an implements directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```
type
  TAncestor = class
    ...
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    ...
end;

type
  TDerived = class(TAncestor)
    ...
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    ...
end;
```

The override of `Size` adds a write specifier to allow the property to be modified. The overrides of `Text` and `Color` change the visibility of the properties from protected to published. The property override of `Color` also specifies that the property should be filed if its value isn't `clBlue`.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always static. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to `MyObject.Value` invokes `Method1` or `Method2`, even though `MyObject` holds an instance of `TDescendant`. But you can cast `MyObject` to `TDescendant` to access the descendant class's properties and their access specifiers.

```
type
  TAncestor = class
    ...
    property Value: Integer read Method1 write Method2;
  end;

  TDescendant = class(TAncestor)
    ...
    property Value: Integer read Method3 write Method4;
  end;

var
  MyObject: TAncestor;
  ...
  MyObject := TDescendant.Create;
```

Class Properties

Class properties can be accessed without an object reference. Class property accessors must themselves be declared as class static methods, or class fields. A class property is declared with the class property keywords. Class properties cannot be published, and cannot have stored or default value definitions.

You can introduce a block of class static fields within a class declaration by using the class var block declaration. All fields declared after class var have static storage attributes. A class var block is terminated by the following:

- 1 Another class var declaration
- 2 A procedure or function (i.e. method) declaration (including class procedures and class functions)
- 3 A property declaration (including class properties)
- 4 A constructor or destructor declaration
- 5 A visibility scope specifier (public, private, protected, published, strict private, and strict protected)

For example:

```
type
  TMyClass = class
    strict private
      class var          // Note fields must be declared as class fields
        FRed: Integer;
        FGreen: Integer;
        FBlue: Integer;
    public              // ends the class var block
      class property Red: Integer read FRed write FRed;
      class property Green: Integer read FGreen write FGreen;
      class property Blue: Integer read FBlue write FBlue;
  end;
```

You can access the above class properties with the code:

```
TMyClass.Red := 0;
TMyClass.Blue := 0;
TMyClass.Green := 0;
```


Events

This topic describes the following material:

- Event properties and event handlers
- Triggering multiple event handlers
- Multicast events (.NET)

About Events

An event links an occurrence in the system with the code that responds to that occurrence. The occurrence triggers the execution of a procedure called an event handler. The event handler performs the tasks that are required in response to the occurrence. Events allow the behavior of a component to be customized at design-time or at runtime. To change the behavior of the component, replace the event handler with a custom event handler that will have the desired behavior.

Event Properties and Event Handlers

Components that are written in Delphi use properties to indicate the event handler that will be executed when the event occurs. By convention, the name of an event property begins with "On", and the property is implemented with a field rather than read/write methods. The value stored by the property is a method pointer, pointing to the event handler procedure.

In the following example, the `TObservedObject` class includes an `OnPing` event, of type `TPingEvent`. The `FOnPing` field is used to store the event handler. The event handler in this example, `TListener.Ping`, prints 'TListener has been pinged!'.

```
Program EventDemo;
{$APPTYPE CONSOLE}

type
  TPingEvent = procedure of object;
  TObservedObject = class
  private
    FPing: TPingEvent;
  public
    property OnPing: TPingEvent read FPing write FPing;
  end;

  TListener = class
    procedure Ping;
  end;

procedure TListener.Ping;
begin
  writeln('TListener has been pinged.');
```

```
end;

var
  observedObject: TObservedObject;
  listener: TListener;

begin
  observedObject := TObservedObject.Create;
  listener := TListener.Create;
```

```

    observedObject.OnPing := listener.Ping;

    observedObject.OnPing; // should output 'TListener has been pinged.'

    ReadLn; // pause console before closing
end.

```

Triggering Multiple Event Handlers

In Delphi for Win32, events can be assigned only a single event handler. If multiple event handlers must be executed in response to an event, the event handler assigned to the event must call any other event handlers. In the following code, a subclass of `TListener` called `TListenerSubclass` has its own event handler called `Ping2`. In this example, the `Ping2` event handler must explicitly call the `TListener.Ping` event handler in order to trigger it in response to the `OnPing` event.

```

Program EventDemo2;
{$APPTYPE CONSOLE}

type
  TPingEvent = procedure of object;
  TObservedObject = class
  private
    FPing: TPingEvent;
  public
    property OnPing: TPingEvent read FPing write FPing;
  end;

  TListener = class
    procedure Ping;
  end;

  TListenerSubclass = class (TListener)
    procedure Ping2;
  end;

procedure TListener.Ping;
begin
    writeln('TListener has been pinged.');
```

```

end;

procedure TListenerSubclass.Ping2;
begin
    self.Ping;
    writeln('TListenerSubclass has been pinged.');
```

```

end;

var
    observedObject: TObservedObject;
    listener: TListenerSubclass;

begin
    observedObject := TObservedObject.Create;
    listener := TListenerSubclass.Create;

```

```

    observedObject.OnPing := listener.Ping2;

    observedObject.OnPing; // should output 'TListener has been pinged.'
    // and then 'TListenerSubclass has been pinged.'

    ReadLn; // pause console before closing
end.

```

Multicast Events (.NET only)

On the .NET platform, Delphi enables multiple event handlers to be applied to the same event. A multicast event is an event that can trigger multiple event handlers. Using multicast events can reduce the effort required for maintenance of complex event systems, and improve the readability and flexibility of event handling.

Multicast events are declared using the `add` and `remove` keywords to indicate the field or methods that are used to add or remove event handlers for an event. The `Include()` and `Exclude()` standard procedures are used to include and exclude event handlers at runtime. The following code demonstrates the declaration of an event property that uses multicast events.

```

Program NETEvents;
{$APPTYPE CONSOLE}

type
    TPingEvent = procedure of object;
    TObservedObject = class
    private
        FPing: TPingEvent;
    public
        property OnPing: TPingEvent add FPing remove FPing;
    end;

    TListener = class
        procedure Ping;
    end;

    TListenerSubclass = class (TListener)
        procedure Ping2;
    end;

procedure TListener.Ping;
begin
    writeln('TListener has been pinged.');
```

```

end;

procedure TListenerSubclass.Ping2;
begin
    writeln('TListenerSubclass has been pinged.');
```

```

end;

var
    observedObject: TObservedObject;
    listener: TListener;
    listenerSubclass: TListenerSubclass;
    testEvent: TPingEvent;

begin
    observedObject := TObservedObject.Create;

```

```

listener := TListener.Create;
listenerSubclass := TListenerSubclass.Create;

Include(observedObject.OnPing, listener.Ping);
Include(observedObject.OnPing, listenerSubclass.Ping2);

// testEvent := observedObject.OnPing;    // not allowed

observedObject.FPing(); // should output 'TListener has been pinged.'

ReadLn; // pause console before closing
end.

```

The `ObservedObject.OnPing` property declaration uses the add and remove keywords instead of read and write. The add and remove keywords indicate to the compiler that `OnClick` is a multicast event.

A property which represents a multicast event can not be read or written directly; it can only be accessed through the `Include()` and `Exclude()` standard procedures. To attempt to read, write or execute a property that has been declared with add and remove is an error at compile time. To execute this event directly, the sample code uses the `FPing` field rather than the `OnPing` property.

Class References

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

This topic covers the following material:

- Class reference types
- Class operators

Class-Reference Types

A class-reference type, sometimes called a metaclass, is denoted by a construction of the form

`class of type`

where *type* is any class type. The identifier *type* itself denotes a value whose type is `class of type`. If *type1* is an ancestor of *type2*, then `class of type2` is assignment-compatible with class of *type1*. Thus

```
type TClass = class of TObject;  
var AnyObj: TClass;
```

declares a variable called `AnyObj` that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value `nil` to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for `TCollection` (in the `Classes` unit):

```
type TCollectionItemClass = class of TCollectionItem;  
...  
constructor Create(ItemClass: TCollectionItemClass);
```

This declaration says that to create a `TCollection` instance object, you must pass to the constructor the name of a class descending from `TCollectionItem`.

Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

Constructors and Class References

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```
type TControlClass = class of TControl;  
  
function CreateControl(ControlClass: TControlClass;  
  const ControlName: string; X, Y, W, H: Integer): TControl;  
begin  
  Result := ControlClass.Create(MainForm);  
  with Result do
```

```

begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
end;
end;

```

The `CreateControl` function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to `CreateControl` can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

Class Operators

Class methods operate on class references. Every class inherits two class methods from `TObject`, called `ClassType` and `ClassParent`. These methods return, respectively, a reference to the class of an object and to an object's immediate ancestor class. Both methods return a value of type `TClass` (where `TClass = class of TObject`), which can be cast to a more specific type. Every class also inherits a method called `InheritsFrom` that tests whether the object where it is called descends from a specified class. These methods are used by the `is` and `as` operators, and it is seldom necessary to call them directly.

The *is* Operator

The `is` operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression

objectisclass

returns `True` if *object* is an instance of the class denoted by *class* or one of its descendants, and `False` otherwise. (If *object* is `nil`, the result is `False`.) If the declared type of *object* is unrelated to *class* - that is, if the types are distinct and one is not an ancestor of the other - a compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to `TEdit` after first verifying that the object it references is an instance of `TEdit` or one of its descendants.

The *as* Operator

The `as` operator performs checked typecasts. The expression

objectasclass

returns a reference to the same object as *object*, but with the type given by *class*. At runtime, *object* must be an instance of the class denoted by *class* or one of its descendants, or be `nil`; otherwise an exception is raised. If the declared type of *object* is unrelated to *class* - that is, if the types are distinct and one is not an ancestor of the other - a compilation error results. For example,

```
with Sender as TButton do  
begin  
  Caption := '&Ok';  
  OnClick := OkClick;  
end;
```

The rules of operator precedence often require as typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```


Exceptions

This topic covers the following material:

- A conceptual overview of exceptions and exception handling
- Declaring exception types
- Raising and handling exceptions

About Exceptions

An exception is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an exception handler, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the `SysUtils` unit, most runtime errors are automatically converted into exceptions. Many errors that would otherwise terminate an application - such as insufficient memory, division by zero, and general protection faults - can be caught and handled.

When To Use Exceptions

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The requirements imposed by exception handling semantics impose a code/data size and runtime performance penalty. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a `try...except` or `try...finally` statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in `if...then` statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
  AssignFile(F, FileName);
  Reset(F);      // raises an EInOutError exception if file is not found
except
  on Exception do ...
end;
```

But you could also avoid the overhead of exception handling by using

```
if FileExists(FileName) then    // returns False if file is not found; raises no exception
```

```
begin
  AssignFile(F, FileName);
```

```
Reset(F);  
end;
```

Assertions provide another way of testing a Boolean condition anywhere in your source code. When an `Assert` statement fails, the program either halts with a runtime error or (if it uses the `SysUtils` unit) raises an `EAssertionFailed` exception. Assertions should be used only to test for conditions that you do not expect to occur.

Declaring Exception Types

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the `Exception` class defined in `SysUtils`.

You can group exceptions into families using inheritance. For example, the following declarations in `SysUtils` define a family of exception types for math errors.

```
type  
  EMathError = class(Exception);  
  EInvalidOp = class(EMathError);  
  EZeroDivide = class(EMathError);  
  EOverflow = class(EMathError);  
  EUnderflow = class(EMathError);
```

Given these declarations, you can define a single `EMathError` exception handler that also handles `EInvalidOp`, `EZeroDivide`, `EOverflow`, and `EUnderflow`.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```
type EInOutError = class(Exception)  
  ErrorCode: Integer;  
end;
```

Raising and Handling Exceptions

To raise an exception object, use an instance of the exception class with a `raise` statement. For example,

```
raise EMathError.Create;
```

In general, the form of a `raise` statement is

`raise object at address`

where `object` and `at address` are both optional. When an address is specified, it can be any expression that evaluates to a pointer type, but is usually a pointer to a procedure or function. For example:

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

Use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is raised - that is, referenced in a `raise` statement - it is governed by special exception-handling logic. A `raise` statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose `try...except` block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an `ERangeError` exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
    Result := StrToInt(S);    // StrToInt is declared in SysUtils
    if (Result < Min) or (Result > Max) then
        raise ERangeError.CreateFmt('%d is not within the valid range of %d..%d', [Result,
Min, Max]);
    end;
```

Notice the *CreateFmt* method called in the raise statement. Exception and its descendants have special constructors that provide alternative ways to create exception messages and context IDs.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

Note: Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the `SysUtils` unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units - including `SysUtils` - are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program. Similarly, raising an exception in the finalization section of a unit may not lead to the intended result if `SysUtils` has already been finalized when the exception has been raised.

Try...except Statements

Exceptions are handled within `try...except` statements. For example,

```
try
    X := Y/Z;
except
    on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide `Y` by `Z`, but calls a routine named `HandleZeroDivide` if an `EZeroDivide` exception is raised.

The syntax of a `try...except` statement is

`try statementsexceptexceptionBlockend`

where *statements* is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

elsestatements

An exception handler has the form

onidentifier: typedostatement

where *identifier* is optional (if included, identifier can be any valid identifier), *type* is a type used to represent exceptions, and *statement* is any statement.

A `try...except` statement executes the statements in the initial statements list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial statements list, either by a raise statement in the statements list or by a procedure or function called from the statements list, an attempt is made to 'handle' the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler 'matches' an exception just in case the type in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the statement in the else clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered `try...except` statement that has not yet exited. If no appropriate handler, else clause, or statement list is found there, the search propagates to the next-most-recently entered `try...except` statement, and so forth. If the outermost `try...except` statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the `try...except` statement where the handling occurs, and control is transferred to the executed exception handler, else clause, or statement list. This process discards all procedure and function calls that occurred after entering the `try...except` statement where the exception is handled. The exception object is then automatically destroyed through a call to its `Destroy` destructor and control is passed to the statement following the `try...except` statement. (If a call to the `Exit`, `Break`, or `Continue` standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. `EMathError` appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows `on...do`. The scope of the identifier is limited to that statement. For example,

```
try
...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an else clause, the else clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
...
except
  on EZeroDivide do HandleZeroDivide;
```

```

    on EOverflow do HandleOverflow;
    on EMathError do HandleMathError;
else
    HandleAllOthers;
end;

```

Here, the `else` clause handles any exception that isn't an `EMathError`.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```

try
    ...
except
    HandleException;
end;

```

Here, the `HandleException` routine handles any exception that occurs as a result of executing the statements between `try` and `except`.

Re-raising Exceptions

When the reserved word `raise` occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then re-raise the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the `GetFileList` function allocates a `TStringList` object and fills it with file names matching a specified search path:

```

function GetFileList(const Path: string): TStringList;
var
    I: Integer;
    SearchRec: TSearchRec;
begin
    Result := TStringList.Create;
    try
        I := FindFirst(Path, 0, SearchRec);
        while I = 0 do
            begin
                Result.Add(SearchRec.Name);
                I := FindNext(SearchRec);
            end;
        except
            Result.Free;
            raise;
        end;
    end;
end;

```

`GetFileList` creates a `TStringList` object, then uses the `FindFirst` and `FindNext` functions (defined in `SysUtils`) to initialize it. If the initialization fails - for example because the search path is invalid, or because there is not enough memory to fill in the string list - `GetFileList` needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a `try...except` statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

Nested Exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the Tan function below.

```
type
  ETrigError = class(EMathError);
function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
    end;
  end;
end;
```

If an EMathError exception occurs during execution of Tan, the exception handler raises an ETrigError. Since Tan does not provide a handler for ETrigError, the exception propagates beyond the original exception handler, causing the EMathError exception to be destroyed. To the caller, it appears as if the Tan function has raised an ETrigError exception.

Try...finally Statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a try...finally statement.

The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```
Reset(F);
try
  ... // process file F
finally
  CloseFile(F);
end;
```

The syntax of a try...finally statement is

`try statementList1 finally statementList2 end`

where each *statementList* is a sequence of statements delimited by semicolons. The try...finally statement executes the statements in *statementList1* (the try clause). If *statementList1* finishes without raising exceptions, *statementList2* (the finally clause) is executed. If an exception is raised during execution of *statementList1*, control is transferred to *statementList2*; once *statementList2* finishes executing, the exception is re-raised. If a call to the Exit, Break, or Continue procedure causes control to leave *statementList1*, *statementList2* is automatically executed. Thus the finally clause is always executed, regardless of how the try clause terminates.

If an exception is raised but not handled in the finally clause, that exception is propagated out of the try...finally statement, and any exception already raised in the try clause is lost. The finally clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

Standard Exception Classes and Routines

The `SysUtils` and `System` units declare several standard routines for handling exceptions, including `ExceptObject`, `ExceptAddr`, and `ShowException`. `SysUtils`, `System` and other units also include dozens of exception classes, all of which (aside from `OutlineError`) derive from `Exception`.

The `Exception` class has properties called `Message` and `HelpContext` that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways.

Nested Type Declarations

Type declarations can be nested within class declarations. Nested types are used throughout the .NET framework, and throughout object-oriented programming in general. They present a way to keep conceptually related types together, and to avoid name collisions. The same syntax for declaring nested types may be used with the Win32 Delphi compiler.

Declaring Nested Types

The *nestedTypeDeclaration* follows the type declaration syntax defined in Declaring Types.

```
type
  className = class [abstract | sealed] (ancestorType)
    memberList

    type
      nestedTypeDeclaration

    memberList
  end;
```

Nested type declarations are terminated by the first occurrence of a non-identifier token, for example, procedure, class, type, and all visibility scope specifiers.

The normal accessibility rules apply to nested types and their containing types. A nested type can access an instance variable (field, property, or method) of its container class, but it must have an object reference to do so. A nested type can access class fields, class properties, and class static methods without an object reference, but the normal Delphi visibility rules apply.

Nested types do not increase the size of the containing class. Creating an instance of the containing class does not also create an instance of a nested type. Nested types are associated with their containing classes only by the context of their declaration.

Declaring and Accessing Nested Classes

The following example demonstrates how to declare and access fields and methods of a nested class.

```
type
  TOuterClass = class
    strict private
      myField: Integer;

    public
      type
        TInnerClass = class
          public
            myInnerField: Integer;
            procedure innerProc;
          end;

      procedure outerProc;
    end;
```

To implement the `innerProc` method of the inner class, you must qualify its name with the name of the outer class. For example

```
procedure TOuterClass.TInnerClass.innerProc;
begin
    ...
end;
```

To access the members of the nested type, use dotted notation as with regular class member access. For example

```
var
    x: TOuterClass;
    y: TOuterClass.TInnerClass;

begin
    x := TOuterClass.Create;
    x.outerProc;
    ...
    y := TOuterClass.TInnerClass.Create;
    y.innerProc;
```

Nested Constants

Constants can be declared in class types in the same manner as nested type sections. Constant sections are terminated by the same tokens as nested type sections, specifically, reserved words or visibility specifiers. Typed constants are not supported, so you cannot declare nested constants of value types, such as `Currency`, or `TDateTime`.

Nested constants can be of any simple type: ordinal, ordinal subranges, enums, strings, and real types.

The following code demonstrates the declaration of nested constants:

```
type
    TMyClass = class
        const
            x = 12;
            y = TMyClass.x + 23;
        procedure Hello;
        private
            const
                s = 'A string constant';
        end;

begin
    writeln(TMyClass.y);    // Writes the value of y, 35.
end.
```

Operator Overloading

This topic describes Delphi's operator methods and how to overload them.

About Operator Overloading

Delphi for .NET and Delphi for Win32 allow certain functions, or "operators" to be overloaded within record declarations. Delphi for .NET also allows overloading within class declarations. The name of the operator function maps to a symbolic representation in source code. For example, the `Add` operator maps to the `+` symbol. The compiler generates a call to the appropriate overload, matching the context (i.e. the return type, and type of parameters used in the call), to the signature of the operator function. The following table shows the Delphi operators that can be overloaded:

Operator	Category	Declaration Signature	Symbol Mapping
<code>Implicit</code>	Conversion	<code>Implicit(a : type) : resultType;</code>	implicit typecast
<code>Explicit</code>	Conversion	<code>Explicit(a: type) : resultType;</code>	explicit typecast
<code>Negative</code>	Unary	<code>Negative(a: type) : resultType;</code>	-
<code>Positive</code>	Unary	<code>Positive(a: type): resultType;</code>	+
<code>Inc</code>	Unary	<code>Inc(a: type) : resultType;</code>	<code>Inc</code>
<code>Dec</code>	Unary	<code>Dec(a: type): resultType</code>	<code>Dec</code>
<code>LogicalNot</code>	Unary	<code>LogicalNot(a: type): resultType;</code>	<code>not</code>
<code>BitwiseNot</code>	Unary	<code>BitwiseNot(a: type): resultType;</code>	<code>not</code>
<code>Trunc</code>	Unary	<code>Trunc(a: type): resultType;</code>	<code>Trunc</code>
<code>Round</code>	Unary	<code>Round(a: type): resultType;</code>	<code>Round</code>
<code>Equal</code>	Comparison	<code>Equal(a: type; b: type) : Boolean;</code>	<code>=</code>
<code>NotEqual</code>	Comparison	<code>NotEqual(a: type; b: type): Boolean;</code>	<code><></code>
<code>GreaterThan</code>	Comparison	<code>GreaterThan(a: type; b: type) Boolean;</code>	<code>></code>
<code>GreaterThanOrEqual</code>	Comparison	<code>GreaterThanOrEqual(a: type; b: type): resultType;</code>	<code>>=</code>
<code>LessThan</code>	Comparison	<code>LessThan(a: type; b: type): resultType;</code>	<code><</code>
<code>LessThanOrEqual</code>	Comparison	<code>LessThanOrEqual(a: type; b: type): resultType;</code>	<code><=</code>
<code>Add</code>	Binary	<code>Add(a: type; b: type): resultType;</code>	<code>+</code>
<code>Subtract</code>	Binary	<code>Subtract(a: type; b: type) : resultType;</code>	<code>-</code>
<code>Multiply</code>	Binary	<code>Multiply(a: type; b: type) : resultType;</code>	<code>*</code>
<code>Divide</code>	Binary	<code>Divide(a: type; b: type) : resultType;</code>	<code>/</code>
<code>IntDivide</code>	Binary	<code>IntDivide(a: type; b: type): resultType;</code>	<code>div</code>
<code>Modulus</code>	Binary	<code>Modulus(a: type; b: type): resultType;</code>	<code>mod</code>
<code>ShiftLeft</code>	Binary	<code>ShiftLeft(a: type; b: type): resultType;</code>	<code>shl</code>
<code>ShiftRight</code>	Binary	<code>ShiftRight(a: type; b: type): resultType;</code>	<code>shr</code>
<code>LogicalAnd</code>	Binary	<code>LogicalAnd(a: type; b: type): resultType;</code>	<code>and</code>
<code>LogicalOr</code>	Binary	<code>LogicalOr(a: type; b: type): resultType;</code>	<code>or</code>
<code>LogicalXor</code>	Binary	<code>LogicalXor(a: type; b: type): resultType;</code>	<code>xor</code>

BitwiseAnd	Binary	BitwiseAnd(a: type; b: type): resultType;	and
BitwiseOr	Binary	BitwiseOr(a: type; b: type): resultType;	or
BitwiseXor	Binary	BitwiseXor(a: type; b: type): resultType;	xor

No operators other than those listed in the table may be defined on a class or record.

Overloaded operator methods cannot be referred to by name in source code. To access a specific operator method of a specific class or record, you must use explicit typecasts on all of the operands. Operator identifiers are not included in the class or record's list of members.

No assumptions are made regarding the distributive or commutative properties of the operation. For binary operators, the first parameter is always the left operand, and the second parameter is always the right operand. Associativity is assumed to be left-to-right in the absence of explicit parentheses.

Resolution of operator methods is done over the union of accessible operators of the types used in the operation (note this includes inherited operators). For an operation involving two different types A and B, if type A has an implicit conversion to B, and B has an implicit conversion to A, an ambiguity will occur. Implicit conversions should be provided only where absolutely necessary, and reflexivity should be avoided. It is best to let type B implicitly convert itself to type A, and let type A have no knowledge of type B (or vice versa).

As a general rule, operators should not modify their operands. Instead, return a new value, constructed by performing the operation on the parameters.

Overloaded operators are used most often in records (i.e. value types). Very few classes in the .NET framework have overloaded operators, but most value types do.

Declaring Operator Overloads

Operator overloads are declared within classes or records, with the following syntax:

```
type
  typeName = [class | record]
    class operator conversionOp(a: type): resultType;
    class operator unaryOp(a: type): resultType;
    class operator comparisonOp(a: type; b: type): Boolean;
    class operator binaryOp(a: type; b: type): resultType;
  end;
```

Implementation of overloaded operators must also include the class operator syntax:

```
class operator typeName.conversionOp(a: type): resultType;
class operator typeName.unaryOp(a: type): resultType;
class operator typeName.comparisonOp(a: type; b: type): Boolean;
class operator typeName.binaryOp(a: type; b: type): resultType;
```

The following are some examples of overloaded operators:

```
type
  TMyClass = class
    class operator Add(a, b: TMyClass): TMyClass;           // Addition of two operands of type
TMyClass
    class operator Subtract(a, b: TMyClass): TMyClass;      // Subtraction of type TMyClass
    class operator Implicit(a: Integer): TMyClass;          // Implicit conversion of an Integer
to type TMyClass
```

```

    class operator Implicit(a: TMyClass): Integer;      // Implicit conversion of TMyClass
to Integer
    class operator Explicit(a: Double): TMyClass;      // Explicit conversion of a Double
to TMyClass
    end;

// Example implementation of Add
class operator TMyClass.Add(a, b: TMyClass): TMyClass;
begin
    // ...
end;

var
x, y: TMyClass;
begin
    x := 12;      // Implicit conversion from an Integer
    y := x + x;   // Calls TMyClass.Add(a, b: TMyClass): TMyClass
    b := b + 100; // Calls TMyClass.Add(b, TMyClass.Implicit(100))
end;

```


Class Helpers

This topic describes the syntax of class helper declarations.

About Class Helpers

A class helper is a type that - when associated with another class - introduces additional method names and properties which may be used in the context of the associated class (or its descendants). Class helpers are a way to extend a class without using inheritance. A class helper simply introduces a wider scope for the compiler to use when resolving identifiers. When you declare a class helper, you state the helper name, and the name of the class you are going to extend with the helper. You can use the class helper any place where you can legally use the extended class. The compiler's resolution scope then becomes the original class, plus the class helper.

Class helpers provide a way to extend a class, but they should not be viewed as a design tool to be used when developing new code. They should be used solely for their intended purpose, which is language and platform RTL binding.

Class Helper Syntax

The syntax for declaring a class helper is:

```
type
  identifierName = class helper [(ancestor list)] for classTypeIdentifierName
    memberList
  end;
```

The *ancestor list* is optional.

A class helper type may not declare instance data, but class fields are allowed.

The visibility scope rules and *memberList* syntax are identical to that of ordinary class types.

You can define and associate multiple class helpers with a single class type. However, only zero or one class helper applies in any specific location in source code. The class helper defined in the nearest scope will apply. Class helper scope is determined in the normal Delphi fashion (i.e. right to left in the unit's uses clause).

Using Class Helpers

The following code demonstrates the declaration of a class helper:

```
type
  TMyClass = class
    procedure MyProc;
    function MyFunc: Integer;
  end;

  ...

  procedure TMyClass.MyProc;
  var X: Integer;
  begin
    X := MyFunc;
  end;
```

```

    function TMyClass.MyFunc: Integer;
    begin
        ...
    end;

...

type
    TMyClassHelper = class helper for TMyClass
        procedure HelloWorld;
        function MyFunc: Integer;
    end;

    ...

    procedure TMyClassHelper.HelloWorld;
    begin
        writeln(Self.ClassName); // Self refers to TMyClass type, not TMyClassHelper
    end;

    function TMyClassHelper.MyFunc: Integer;
    begin
        ...
    end;

...

var
    X: TMyClass;
begin
    X := TMyClass.Create;
    X.MyProc;    // Calls TMyClass.MyProc
    X.HelloWorld; // Calls TMyClassHelper.HelloWorld
    X.MyFunc;    // Calls TMyClassHelper.MyFunc
end;

```

Note that the class helper function `MyFunc` is called, since the class helper takes precedence over the actual class type.

Standard Routines and I/O

This section describes the standard routines included in the Delphi runtime library.

In This Section

[Standard Routines and I/O](#)

Describes text and file I/O and standard library routines.

Standard Routines and I/O

These topics discuss text and file I/O and summarize standard library routines. Many of the procedures and functions listed here are defined in the [System](#) and [SysInit](#) units, which are implicitly used with every application. Others are built into the compiler but are treated as if they were in the [System](#) unit.

Some standard routines are in units such as [SysUtils](#), which must be listed in a uses clause to make them available in programs. You cannot, however, list [System](#) in a uses clause, nor should you modify the [System](#) unit or try to rebuild it explicitly.

File Input and Output

The table below lists input and output routines.

Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
ReadLn	Does what Read does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.

A file variable is any variable whose type is a file type. There are three classes of file: typed, text, and untyped. The syntax for declaring file types is given in File types. Note that file types are only available on the Win32 platform.

Before a file variable can be used, it must be associated with an external file through a call to the AssignFile procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be *opened* to prepare it for input or output. An existing file can be opened via the Reset procedure, and a new file can be created and opened via the Rewrite procedure. Text files opened with Reset are read-only and text files opened with Rewrite and Append are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with Reset or Rewrite.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure Read or written using the standard procedure Write, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure Seek, which moves the current file position to a specified component. The standard functions FilePos and FileSize can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure CloseFile. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{SI+}** and **{SI-}** compiler directives. When I/O checking is off, that is, when a procedure or function call is compiled in the **{SI-}** state an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function IOResult instead.

You must call the IOResult function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{SI-}** is the current state, the next I/O function call will fail with the lingering IOResult error.

Text Files

This section summarizes I/O using file variables of the standard type Text.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a line feed character). The type Text is distinct from the type file of Char.

For text files, there are special forms of Read and Write that let you read and write values that are not of type Char. Such values are automatically translated to and from their character representation. For example, `Read(F, I)`, where I is a type Integer variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in I.

There are two standard text file variables, Input and Output. The standard file variable Input is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable Output is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, Input and Output are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');  
Reset(Input);  
AssignFile(Output, '');  
Rewrite(Output);
```

Note: For Win32 applications, text-oriented I/O is available only in console applications, that is, applications compiled with the Generate console application option checked on the Linker page of the Project Options dialog box or with the `-cc` command-line compiler option. In a GUI (non-console) application, any attempt to read or write using `Input` or `Output` will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, `Input` or `Output` is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, `Read(X)` corresponds to `Read(Input, X)` and `Write(X)` corresponds to `Write(Output, X)`.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using `AssignFile`, and opened using `Reset`, `Rewrite`, or `Append`. An error occurs if you pass a file that was opened with `Reset` to an output-oriented procedure or function. An error also occurs if you pass a file that was opened with `Rewrite` or `Append` to an input-oriented procedure or function.

Untyped Files

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word `file` and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the `Reset` and `Rewrite` procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for `Read` and `Write`, all typed-file standard procedures and functions are also allowed on untyped files. Instead of `Read` and `Write`, two procedures called `BlockRead` and `BlockWrite` are used for high-speed data transfers.

Text File Device Drivers

You can define your own text file device drivers for your programs. A text file device driver is a set of four functions that completely implement an interface between Delphi's file system and some device.

The four functions that define each device driver are `Open`, `InOut`, `Flush`, and `Close`. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where `DeviceFunc` is the name of the function (that is, `Open`, `InOut`, `Flush`, or `Close`). The return value of a device-interface function becomes the value returned by `IOResult`. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized `Assign` procedure. The `Assign` procedure must assign the addresses of the four device-interface functions to the four function pointers in the text file variable. In addition, it should store the `fmClosedmagic` constant in the `Mode` field, store the size of the text file buffer in `BufSize`, store a pointer to the text file buffer in `BufPtr`, and clear the `Name` string.

Assuming, for example, that the four device-interface functions are called `DevOpen`, `DevInOut`, `DevFlush`, and `DevClose`, the `Assign` procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device-interface functions can use the `UserData` field in the file record to store private information. This field isn't modified by the product file system at any time.

The Open function

The `Open` function is called by the `Reset`, `Rewrite`, and `Append` standard procedures to open a text file associated with a device. On entry, the `Mode` field contains `fmInput`, `fmOutput`, or `fmInOut` to indicate whether the `Open` function was called from `Reset`, `Rewrite`, or `Append`.

The `Open` function prepares the file for input or output, according to the `Mode` value. If `Mode` specified `fmInOut` (indicating that `Open` was called from `Append`), it must be changed to `fmOutput` before `Open` returns.

`Open` is always called before any of the other device-interface functions. For that reason, `AssignDev` only initializes the `OpenFunc` field, leaving initialization of the remaining vectors up to `Open`. Based on `Mode`, `Open` can then install pointers to either input- or output-oriented functions. This saves the `InOut`, `Flush` functions and the `CloseFile` procedure from determining the current mode.

The InOut function

The `InOut` function is called by the `Read`, `Readln`, `Write`, `Writeln`, `Eof`, `Eoln`, `SeekEof`, `SeekEoln`, and `CloseFile` standard routines whenever input or output from the device is required.

When `Mode` is `fmInput`, the `InOut` function reads up to `BufSize` characters into `BufPtr^`, and returns the number of characters read in `BufEnd`. In addition, it stores zero in `BufPos`. If the `InOut` function returns zero in `BufEnd` as a result of an input request, `Eof` becomes **True** for the file.

When `Mode` is `fmOutput`, the `InOut` function writes `BufPos` characters from `BufPtr^`, and returns zero in `BufPos`.

The Flush function

The `Flush` function is called at the end of each `Read`, `Readln`, `Write`, and `Writeln`. It can optionally flush the text file buffer.

If `Mode` is `fmInput`, the `Flush` function can store zero in `BufPos` and `BufEnd` to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If `Mode` is `fmOutput`, the `Flush` function can write the contents of the buffer exactly like the `InOut` function, which ensures that text written to the device appears on the device immediately. If `Flush` does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The `Close` function is called by the `CloseFile` standard procedure to close a text file associated with a device. (The `Reset`, `Rewrite`, and `Append` procedures also call `Close` if the file they are opening is already open.) If `Mode` is `fmOutput`, then before calling `Close`, the file system calls the `InOut` function to ensure that all characters have been written to the device.

Handling null-Terminated Strings

The Delphi language's extended syntax allows the `Read`, `Readln`, `Str`, and `Val` standard procedures to be applied to zero-based character arrays, and allows the `Write`, `Writeln`, `Val`, `AssignFile`, and `Rename` standard procedures to be applied to both zero-based character arrays and character pointers.

Null-Terminated String Functions

The following functions are provided for handling null-terminated strings.

Null-terminated string functions

Function	Description
<code>StrAlloc</code>	Allocates a character buffer of a given size on the heap.
<code>StrBufSize</code>	Returns the size of a character buffer allocated using <code>StrAlloc</code> or <code>StrNew</code> .
<code>StrCat</code>	Concatenates two strings.
<code>StrComp</code>	Compares two strings.
<code>StrCopy</code>	Copies a string.
<code>StrDispose</code>	Disposes a character buffer allocated using <code>StrAlloc</code> or <code>StrNew</code> .
<code>StrECopy</code>	Copies a string and returns a pointer to the end of the string.
<code>StrEnd</code>	Returns a pointer to the end of a string.
<code>StrFmt</code>	Formats one or more values into a string.
<code>StrIComp</code>	Compares two strings without case sensitivity.
<code>StrLCat</code>	Concatenates two strings with a given maximum length of the resulting string.
<code>StrLComp</code>	Compares two strings for a given maximum length.
<code>StrLCopy</code>	Copies a string up to a given maximum length.
<code>StrLen</code>	Returns the length of a string.
<code>StrLFmt</code>	Formats one or more values into a string with a given maximum length.
<code>StrLIComp</code>	Compares two strings for a given maximum length without case sensitivity.
<code>StrLower</code>	Converts a string to lowercase.
<code>StrMove</code>	Moves a block of characters from one string to another.
<code>StrNew</code>	Allocates a string on the heap.
<code>StrPCopy</code>	Copies a Pascal string to a null-terminated string.
<code>StrPLCopy</code>	Copies a Pascal string to a null-terminated string with a given maximum length.
<code>StrPos</code>	Returns a pointer to the first occurrence of a given substring within a string.
<code>StrRScan</code>	Returns a pointer to the last occurrence of a given character within a string.
<code>StrScan</code>	Returns a pointer to the first occurrence of a given character within a string.

StrUpper	Converts a string to uppercase.
----------	---------------------------------

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with Ansi-. For example, the multibyte version of StrPos is AnsiStrPos. Multibyte character support is operating-system dependent and based on the current locale.

Wide-Character Strings

The `System` unit provides three functions, `WideCharToString`, `WideCharLenToString`, and `StringToWideChar`, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

Assignment will also convert between strings. For instance, the following are both valid:

```
MyAnsiString := MyWideString;  
MyWideString := MyAnsiString;
```

Other Standard Routines

The table below lists frequently used procedures and functions found in Borland product libraries. This is not an exhaustive inventory of standard routines.

Other standard routines

Procedure or function	Description
Addr	Returns a pointer to a specified object.
AllocMem	Allocates a memory block and initializes each byte to zero.
ArcTan	Calculates the arctangent of the given number.
Assert	Raises an exception if the passed expression does not evaluate to true.
Assigned	Tests for a nil (unassigned) pointer or procedural variable.
Beep	Generates a standard beep.
Break	Causes control to exit a for, while, or repeat statement.
ByteToCharIndex	Returns the position of the character containing a specified byte in a string.
Chr	Returns the character for a specified integer value.
Close	Closes a file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of for, while, or repeat statements.
Copy	Returns a substring of a string or a segment of a dynamic array.
Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type <code>TDateTime</code> to a string.
DateToStr	Converts a variable of type <code>TDateTime</code> to a string.
Dec	Decrements an ordinal variable or a typed pointer variable.

Dispose	Releases dynamically allocated variable memory.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.
Finalize	Finalizes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Releases allocated memory.
GetMem	Allocates dynamic memory and a pointer to the address of the block.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.
High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable or a typed pointer variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.
Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a dynamic allocated variable memory and references it with a specified pointer.
Now	Returns the current date and time.
Ord	Returns the ordinal integer value of an ordinal-type expression.
Pos	Returns the index of the first single-byte character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a value to a pointer.
Random	Generates random numbers within a specified range.

ReallocMem	Reallocates a dynamically allocatable memory.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.
ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Converts an integer or real number into a string.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (TDateTime).
StrToDateTime	Converts a string to a TDateTime.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.
StrToTime	Converts a string to a time format (TDateTime).
StrUpper	Returns an ASCII string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type TDateTime to a string.
Trunc	Truncates a real number to an integer.
UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a variant array.
VarArrayDimCount	Returns number of dimensions of a variant array.
VarArrayHighBound	Returns high bound for a dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a variant array.
VarArrayOf	Creates and fills a one-dimensional variant array.
VarArrayRedim	Resizes a variant array.
VarArrayRef	Returns a reference to the passed variant array.
VarArrayUnlock	Unlocks a variant array.
VarAsType	Converts a variant to specified type.
VarCast	Converts a variant to a specified type, storing the result in a variable.

VarClear	Clears a variant.
VarCopy	Copies a variant.
VarToStr	Converts variant to string.
VarType	Returns type code of specified variant.

Libraries and Packages

This section describes how to create static and dynamically loadable libraries in Delphi.

In This Section

[Libraries and Packages](#)

Describes the use of libraries and packages.

[Writing Dynamically Loaded Libraries](#)

Describes the issues involved in writing Dynamically Loaded Libraries.

[Packages](#)

Describes packages and how to implement them.

Libraries and Packages

A dynamically loadable library is a dynamic-link library (DLL) on Win32, and an assembly (also a DLL) on the .NET platform. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

Delphi programs can call DLLs and assemblies written in other languages, and applications written in other languages can call DLLs or assemblies written in Delphi.

Calling Dynamically Loadable Libraries

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in DLL or assembly, you must import them. This can be done in two ways: by declaring an external procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

The Delphi language does not support importing of variables from DLLs or assemblies.

Static Loading

The simplest way to import a procedure or function is to declare it using the external directive. For example,

```
procedure DoSomething; external 'MYLIB.DLL';
```

If you include this declaration in a program, MYLIB.DLL is loaded once, when the program starts. Throughout execution of the program, the identifier `DoSomething` always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect external declarations into a separate "import unit" that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

Dynamic Loading

You can access routines in a library through direct calls to Win32 APIs, including `LoadLibrary`, `FreeLibrary`, and `GetProcAddress`. These functions are declared in `Windows.pas`. On Linux, they are implemented for compatibility in `SysUtils.pas`; the actual Linux OS routines are `dlopen`, `dclose`, and `dlsym` (all declared in `libc`; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example,

```
uses Windows, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
```

```

TGetTime = procedure(var Time: TTimeRec);
THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  .
  .
  .
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
      end;
      FreeLibrary(Handle);
    end;
  end;
end;

```

When you import routines this way, the library is not loaded until the code containing the call to [LoadLibrary](#) executes. The library is later unloaded by the call to [FreeLibrary](#). This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.

Writing Dynamically Loaded Libraries

The following topics describe elements of writing dynamically loadable libraries, including

- The exports clause.
- Library initialization code.
- Global variables.
- Libraries and system variables.

Using Export Clause in Libraries

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word `library` (instead of `program`).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, `Min` and `Max`.

```
library MinMax;
  function Min(X, Y: Integer): Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;
  function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;
  exports
    Min,
    Max;
begin
end.
```

If you want your library to be available to applications written in other languages, it's safest to specify `stdcall` in the declarations of exported functions. Other languages may not support Delphi's default register calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a `uses` clause, an `exports` clause, and the initialization code. For example,

```

library Editors;
  uses EdInit, EdInOut, EdFormat, EdPrint;
  exports
    InitEditors,
    DoneEditors name Done,
    InsertText name Insert,
    DeleteSelection name Delete,
    FormatSelection,
    PrintSelection name Print,
    .
    .
    .
  SetErrorHandler;
begin
  InitLibrary;
end.

```

You can put exports clauses in the interface or implementation section of a unit. Any library that includes such a unit in its uses clause automatically exports the routines listed the unit's exports clauses without the need for an exports clause of its own.

The directive `local`, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the `local` directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for stand-alone procedures and functions, but not for methods. A routine declared with `local` for example,

```
function Contraband(I: Integer): Integer; local;
```

does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the interface section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the caller sets up EBX.

A routine is exported when it is listed in an exports clause, which has the form

```
exports entry1, ..., entryn;
```

where each entry consists of the name of a procedure, function, or variable (which must be declared prior to the exports clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional name specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive `resident`, which is maintained for backward compatibility and is ignored by the compiler.)

On the Win32 platform, an index specifier consists of the directive `index` followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no index specifier, the routine is automatically assigned a number in the export table.

Note: Use of index specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A name specifier consists of the directive name followed by a string constant. If an entry has no name specifier, the routine is exported under its original declared name, with the same spelling and case. Use a name clause when you want to export a routine under a different name. For example,

```
exports
DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the exports clause. For example,

```
exports
Divide(X, Y: Integer) name 'Divide_Ints',
Divide(X, Y: Real) name 'Divide_Reals';
```

On Win32, do not include index specifiers in entries for overloaded routines.

An exports clause can appear anywhere and any number of times in the declaration part of a program or library, or in the interface or implementation section of a unit. Programs seldom contain an exports clause.

Library Initialization Code

The statements in a library's block constitute the library's initialization code. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an entry point procedure using the `DllProc` variable. The `DllProc` variable is similar to an exit procedure, which is described in Exit procedures; the entry point procedure executes when the library is loaded or unloaded.

Library initialization code can signal an error by setting the `ExitCode` variable to a nonzero value. `ExitCode` is declared in the `System` unit and defaults to zero, indicating successful initialization. If a library's initialization code sets `ExitCode` to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an entry point procedure.

```
library Test;
var
  SaveDllProc: Pointer;
  procedure LibExit(Reason: Integer);
begin
  if Reason = DLL_PROCESS_DETACH then
  begin
    .
    . // library exit code
    .
  end;
  SaveDllProc(Reason); // call saved entry point procedure
end;
begin
  .
  . // library initialization code
  .
  SaveDllProc := DllProc; // save exit procedure chain
  DllProc := @LibExit;    // install LibExit exit procedure
end.
```

`DllProc` is called when the library is first loaded into memory, when a thread starts or stops, or when the library is unloaded. The initialization parts of all units used by a library are executed before the library's initialization code, and the finalization parts of those units are executed after the library's entry point procedure.

Global Variables in a Library

Global variables declared in a shared library cannot be imported by a Delphi application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries - or multiple instances of a library - to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

Libraries and System Variables

Several variables declared in the `System` unit are of special interest to those programming libraries. Use `IsLibrary` to determine whether code is executing in an application or in a library; `IsLibrary` is always `False` in an application and `True` in a library. During a library's lifetime, `HInstance` contains its instance handle. `CmdLine` is always `nil` in a library.

The `DLLProc` variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. `DLLProc` is available on both Windows and Linux but its use differs on each. On Win32, `DLLProc` is used in multithreading applications.; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior.

To monitor operating-system calls, create a callback procedure that takes a single integer parameter for example,

```
procedure DLLHandler(Reason: Integer);
```

and assign the address of the procedure to the `DLLProc` variable. When the procedure is called, it passes to it one of the following values.

<code>DLL_PROCESS_DETACH</code>	Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to <code>FreeLibrary</code> .
<code>DLL_PROCESS_ATTACH</code>	Indicates that the library is attaching to the address space of the calling process as the result of a call to <code>LoadLibrary</code> .
<code>DLL_THREAD_ATTACH</code>	Indicates that the current process is creating a new thread.
<code>DLL_THREAD_DETACH</code>	Indicates that a thread is exiting cleanly.

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

Exceptions and Runtime Errors in Libraries

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Delphi, the exception can be handled through a normal `try...except` statement.

On Win32, if the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code `$0EEDFADE`. The first entry in the `ExceptionInformation` array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Delphi exception object.

Generally, you should not let exceptions escape from your library. Delphi exceptions map to the OS exception model (including the .NET exception model)..

If a library does not use the [SysUtils](#) unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates. Because the library has no way of knowing whether it was called from a Delphi program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

Shared-Memory Manager (Win32 Only)

On Win32, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the [ShareMem](#) unit. The same is true if one application or DLL allocates memory with [New](#) or [GetMem](#) which is deallocated by a call to [Dispose](#) or [FreeMem](#) in another module. [ShareMem](#) should always be the first unit listed in any program or library uses clause where it occurs.

[ShareMem](#) is the interface unit for the BORLANDMM.DLL memory manager, which allows modules to share dynamically allocated memory. BORLANDMM.DLL must be deployed with applications and DLLs that use [ShareMem](#). When an application or DLL uses [ShareMem](#), its memory manager is replaced by the memory manager in BORLANDMM.DLL.

Packages

The following topics describe packages and various issues involved in creating and compiling them.

- Package declarations and source files
- Naming packages
- The requires clause
- Avoiding circular package references
- Duplicate package references
- The contains clause
- Avoiding redundant source code uses
- Compiling packages
- Generated files
- Package-specific compiler directives
- Package-specific command-line compiler switches

Understanding Packages

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange where code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

Runtime packages provide functionality when a user runs an application. Design-time packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their requires clauses.

On Win32, package files end with the .bpl (Borland package library) extension. On the .NET platform, packages are .NET assemblies, and end with an extension of .dll

Ordinarily, packages are loaded statically when an application starts. But you can use the [LoadPackage](#) and [UnloadPackage](#) routines (in the [SysUtils](#) unit) to load packages dynamically.

Note: When an application utilizes packages, the name of each packaged unit still must appear in the uses clause of any source file that references it.

Package Declarations and Source Files

Each package is declared in a separate source file, which should be saved with the .dpk extension to avoid confusion with other files containing Delphi code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains:

- a name for the package.
- a list of other packages required by the new package. These are packages to which the new package is linked.
- a list of unit files contained by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

```
package packageName;  
requiresClause;
```

containsClause;

end.

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the `DATAx` package.

```
package DATAx;  
  requires  
    rtl,  
    contains Db, DBLocal, DBXpress, ... ;  
end.
```

The *requires* clause lists other, external packages used by the package being declared. It consists of the directive *requires*, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a *requires* clause.

The *contains* clause identifies the unit files to be compiled and bound into the package. It consists of the directive *contains*, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word *in* and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

Note: Thread-local variables (declared with *threadvar*) in a packaged unit cannot be accessed from clients that use the package.

Naming packages

A compiled package involves several generated files. For example, the source file for the package called `DATAx` is `DATAx.DPK`, from which the compiler generates an executable and a binary image called

`DATAx.BPL` (Win32) or `DATAx.DLL` (.NET), and `DATAx.DCP` (Win32) or `DATAx.DCPIL` (.NET)

`DATAx` is used to refer to the package in the *requires* clauses of other packages, or when using the package in an application. Package names must be unique within a project.

The requires clause

The *requires* clause lists other, external packages that are used by the current package. It functions like the *uses* clause in a unit file. An external package listed in the *requires* clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's *requires* clause. If the other packages are omitted from the *requires* clause, the compiler loads the referenced units from their `.dcu` or `.dcuil` files.

Avoiding circular package references

Packages cannot contain circular references in their *requires* clauses. This means that

- A package cannot reference itself in its own *requires* clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Duplicate package references

The compiler ignores duplicate references in a package's requires clause. For programming clarity and readability, however, duplicate references should be removed.

The contains clause

The contains clause identifies the unit files to be bound into the package. Do not include file-name extensions in the contains clause.

Avoiding redundant source code uses

A package cannot be listed in the contains clause of another package or the uses clause of a unit.

All units included directly in a package's contains clause, or indirectly in the uses clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in requires clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

Compiling Packages

Packages are ordinarily compiled from the IDE using .dpk files generated by the **Project Manager**. You can also compile .dpk files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

Generated Files

The following table lists the files produced by the successful compilation of a package.

Compiled package files

File extension	Contents
DCP (Win32) or DCPIL (.NET)	A binary image containing a package header and the concatenation of all .dcu (Win32) or .dcuil (.NET) files in the package. A single .dcp or .dcpil file is created for each package. The base name for the file is the base name of the .dpk source file.
BPL (Win32) or DLL (.NET)	The runtime package. This file is a DLL on Win32 with special Borland-specific features. The base name for the package is the base name of the dpk source file.

Package-Specific Compiler Directives

The following table lists package-specific compiler directives that can be inserted into source code.

Package-specific compiler directives

Directive	Purpose
<code>{\$IMPLICITBUILD OFF}</code>	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
<code>{\$G-}</code> or <code>{\$IMPORTEDDATA OFF}</code>	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
<code>{\$WEAKPACKAGEUNIT ON}</code>	Packages unit weakly.
<code>{\$DENYPACKAGEUNIT ON}</code>	Prevents unit from being placed in a package.

<code>{ \$DESIGNONLY ON }</code>	Compiles the package for installation in the IDE. (Put in .dpk file.)
<code>{ \$RUNONLY ON }</code>	Compiles the package as runtime only. (Put in .dpk file.)

Including `{ $DENYPACKAGEUNIT ON }` in source code prevents the unit file from being packaged. Including `{ $G- }` or `{ $IMPORTEDDATA OFF }` may prevent a package from being used in the same application with other packages.

Other compiler directives may be included, if appropriate, in package source code.

Package-Specific Command-Line Compiler Switches

The following package-specific switches are available for the command-line compiler.

Package-specific command-line compiler switches

Switch	Purpose
<code>-\$G-</code>	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
LE path	Specifies the directory where the compiled package file will be placed.
LN path	Specifies the directory where the package dcp or dcpil file will be placed.
LUpackageName [:packageName2;...]	Specifies additional runtime packages to use in an application. Used when compiling a project.
Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the `-$G-` switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

Note: When using the -LU switch on the .NET platform, you can refer to the package with or without the .dll extension. If you omit the .dll extension, the compiler will look for the package on the unit search path, and on the package search path. However, if the package specification contains a drive letter or the path separator character, then the compiler will assume the package name is the full file name (including the .dll extension). In the latter case, if you specify a full or relative path, but omit the .dll extension, the compiler will not be able to locate the package.

Object Interfaces

This section describes the use of interfaces in Delphi.

In This Section

[Object Interfaces](#)

Describes the declaration and implementation of interfaces in Delphi.

[Implementing Interfaces](#)

Describes implementing interface methods.

[Interface References](#)

Describes the declaration and use of interface variables.

Object Interfaces

An object interface, or simply interface, defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models (such as CORBA and SOAP). Using a distributed object model, custom objects that support interfaces can interact with objects written in C++, Java, and other languages.

Interface Types

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
    ['{GUID}']
    memberList
end;
```

where (*ancestorInterface*) and ['{GUID}'] are optional for .NET interfaces.

Warning: Though the ancestor interface and GUID specification are optional for .NET interfaces, they are required to support Win32 COM interoperability. If your interface is to be accessed through COM, be sure to specify the ancestor interface and GUID.

In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property read and write specifiers must be methods.
- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as default.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as virtual, dynamic, abstract, or override. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
end;
```

In some interface declarations, the interface reserved word is replaced by `dispinterface`. This construction (along with the `dispid`, `readonly`, and `writeonly` directives) is platform-specific and is not used in Linux programming.

Interface and Inheritance

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not implement methods. What an interface inherits is the obligation to implement methods, an obligation that is passed onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of `IInterface`, which is defined in the `System` unit and is the ultimate ancestor of all other interfaces. On Win32, `IInterface` declares three methods: `QueryInterface`, `_AddRef`, and `_Release`. These methods are not present on the .NET platform, and you do not need to implement them.

Note: `IInterface` is equivalent to `IUnknown`. You should generally use `IInterface` for platform independent applications and reserve the use of `IUnknown` for specific programs that include Win32 dependencies.

`QueryInterface` provides the means to obtain a reference to the different interfaces that an object supports. `_AddRef` and `_Release` provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the `System` unit's `TInterfacedObject`. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects, however, must be managed through `_AddRef` and `_Release`.

Warning: Though `QueryInterface`, `_AddRef`, and `_Release` are optional for .NET interfaces, they are required to support Win32 COM interoperability. If your interface is to be accessed through COM, be sure to implement these methods.

Interface Identification

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

```
[ '{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}' ]
```

where each x is a hexadecimal digit (0 through 9 or A through F). The Type Library editor automatically generates GUIDs for new interfaces. You can also generate GUIDs by pressing `Ctrl+Shift+G` in the code editor.

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations.

Note: GUIDs are not required for interfaces in the .NET framework. They are only used for COM interoperability. The `TGUID` and `PGUID` types, declared in the `System` unit, are used to manipulate GUIDs.

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

On the .NET platform, you can tag an interface as described above (i.e. following the interface declaration). However, if you use the traditional Delphi syntax, the first square bracket construct following the interface declaration is taken

as a GUID specifier - not as a .NET attribute. (Note that .NET attributes always apply to the *next* symbol, not the previous one.) You can also associate a GUID with an interface using the .NET Guid custom attribute. In this case you would use the .NET style syntax, placing the attribute immediately before the interface declaration.

When you declare a typed constant of type TGUID, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type TGUID. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports can be called in either of two ways

```
if Supports(Allocator, IMalloc) then ...
```

or

```
if Supports(Allocator, IID_IMalloc) then ...
```

Calling Conventions for Interfaces

The default calling convention for interface methods is register, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with stdcall. Use safecall to implement CORBA interfaces. On Win32, you can use safecall to implement methods of dual interfaces.

Interface Properties

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled.

In an interface, property read and write specifiers must be methods, since fields are not available.

Forward Declarations

An interface declaration that ends with the reserved word `interface` and a semicolon, without specifying an ancestor, GUID, or member list, is a forward declaration. A forward declaration must be resolved by a defining declaration of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    .
    .
    .
```

```
end;  
IControl = interface  
['{00000115-0000-0000-C000-000000000049}']  
function GetWindow: IWindow;  
.  
    .  
    .  
end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive `IWindow` from `IControl` and also derive `IControl` from `IWindow`.

Implementing Interfaces

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor.

Class Declarations

Such declarations have the form

```
type className = class (ancestorClass, interfacer1, ..., interfacerN)
    memberList
end;
```

For example,

```
type
    TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    .
    .
    .
end;
```

declares a class called `TMemoryManager` that implements the `IMalloc` and `IErrorInfo` interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the (Win32) declaration of `TInterfacedObject` in the `System` unit. On the .NET platform, `TInterfacedObject` is an alias for `TObject`.

```
type
    TInterfacedObject = class(TObject, IInterface)
    protected
        FRefCount: Integer;
        function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
    public
        procedure AfterConstruction; override;
        procedure BeforeDestruction; override;
        class function NewInstance: TObject; override;
        property RefCount: Integer read FRefCount;
    end;
```

`TInterfacedObject` implements the `IInterface` interface. Hence `TInterfacedObject` declares and implements each of the three `IInterface` methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares `TMemoryManager` as a direct descendent of `TInterfacedObject`.) On the Win32 platform, every interface inherits from `IInterface`, and a class that implements interfaces must implement the `QueryInterface`, `_AddRef`, and `_Release` methods. The `System` unit's `TInterfacedObject` implements these methods and is thus a convenient base from which to derive other classes that implement interfaces. On the .NET platform, `IInterface` does not declare these methods, and you do not need to implement them.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters

in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

Method Resolution Clause

You can override the default name-based mappings by including method resolution clauses in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

```
procedure interface.interfaceMethod = implementingMethod;
```

or

```
function interface.interfaceMethod = implementingMethod;
```

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    .
    .
    .
end;
```

maps *IMalloc*'s *Alloc* and *Free* methods onto *TMemoryManager*'s *Allocate* and *Deallocate* methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

Changing Inherited Implementations

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class' declaration. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    .
    .
    .
end;
TWindow = class(TInterfacedObject, IWindow) // TWindow implements IWindow
```

```

    procedure Draw;
    .
    .
    .
end;
TFrameWindow = class(TWindow, IWindow) // TFrameWindow reimplements IWindow
    procedure Draw;
    .
    .
    .
end;

```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

Implementing Interfaces by Delegation (Win32 only)

On the Win32 platform, the **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

declares a property called `MyInterface` that implements the interface `IMyInterface`.

The implements directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a read specifier. If the property uses a read method, that method must use the default register calling convention and cannot be dynamic (though it can be virtual) or specify the message directive.

The class you use to implement the delegated interface should derive from `TAggregationObject`.

Note: Due to restrictions imposed by the CLR, the implements directive is not supported on the .NET platform.

Delegating to an Interface-Type Property (Win32 only)

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the implements directive, and which does so without method resolution clauses. For example,

```

type
    IMyInterface = interface
        procedure P1;
        procedure P2;
    end;
TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
end;
var
    MyClass: TMyClass;

```

```

MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...// some object whose class implements IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;

```

Delegating to a Class-Type Property (Win32 only)

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```

type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyImplClass = class
    procedure P1;
    procedure P2;
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  end;
  procedure TMyImplClass.P1;
  .
  .
  .
  procedure TMyImplClass.P2;
  .
  .
  .
  procedure TMyClass.MyP1;
  .
  .
  .
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;    // calls TMyClass.MyP1;
  MyInterface.P2;    // calls TImplClass.P2;
end;

```

Interface References

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. These topics describe Interface references and related topics.

Implementing Interface References

Interface reference variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following:

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```
type
    IAncestor = interface
end;
IDescendant = interface(IAncestor)
    procedure P1;
end;
TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
end;
.
.
.
var
    D: IDescendant;
    A: IAncestor;
begin
    D := TSomething.Create; // works!
    A := TSomething.Create; // error
    D.P1; // works!
    D.P2; // error
end;
```

In this example, A is declared as a variable of type `IAncestor`. Because `TSomething` does not list `IAncestor` among the interfaces it implements, a `TSomething` instance cannot be assigned to A. But if we changed `TSomething`'s declaration to

```
TSomething = class(TInterfacedObject, IAncestor, IDescendant)
.
.
.
```

the first error would become a valid assignment. D is declared as a variable of type `IDescendant`. While D references an instance of `TSomething`, we cannot use it to access `TSomething`'s P2 method, since P2 is not a method of `IDescendant`. But if we changed D's declaration to

```
D: TSomething;
```

the second error would become a valid method call.

On the Win32 platform, interface references are typically managed through reference-counting, which depends on the `_AddRef` and `_Release` methods inherited from `IInterface`. These methods, and reference counting in general, are not applicable on the .NET platform, which is a garbage collected environment. Using the default implementation of reference counting, when an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope. Some classes implement interfaces to bypass this default lifetime management, and some hybrid objects use reference counting only when the object does not have an owner.

Global interface-type variables can be initialized only to `nil`.

To determine whether an interface-type expression references an object, pass it to the standard function `Assigned`.

Interface Assignment Compatibility

Variables of a given class type are assignment-compatible with any interface type implemented by the class. Variables of an interface type are assignment-compatible with any ancestor interface type. The value `nil` can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type `IDispatch` or a descendant, the variant receives the type code `varDispatch`. Otherwise, the variant receives the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be assigned to an `IInterface` variable. A variant whose type code is `varEmpty` or `varDispatch` can be assigned to an `IDispatch` variable.

Interface Typecasts

An interface-type expression can be cast to `Variant`. If the interface is of type `IDispatch` or a descendant, the resulting variant has the type code `varDispatch`. Otherwise, the resulting variant has the type code `varUnknown`.

A variant whose type code is `varEmpty`, `varUnknown`, or `varDispatch` can be cast to `IInterface`. A variant whose type code is `varEmpty` or `varDispatch` can be cast to `IDispatch`.

Interface Querying

You can use the `as` operator to perform checked interface typecasts. This is known as interface querying, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

```
object as interface
```

where `object` is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and `interface` is any interface declared with a GUID.

An interface query returns `nil` if `object` is `nil`. Otherwise, it passes the GUID of `interface` to the `QueryInterface` method in `object`, raising an exception unless `QueryInterface` returns zero. If `QueryInterface` returns zero (indicating that `object`'s class implements `interface`), the interface query returns an interface reference to `object`.

Automation Objects (Win32 Only)

An object whose class implements the IDispatch interface (declared in the [System](#) unit) is an Automation object.

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include [ComObj](#) in the uses clause of one of your units or your program or library.

Dispatch Interface Types

Dispatch interface types define the methods and properties that an Automation object implements through IDispatch. Calls to methods of a dispatch interface are routed through IDispatch's Invoke method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

where ['{GUID}'] is optional and memberList consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
end;
```

Dispatch interface methods

Methods of a dispatch interface are prototypes for calls to the Invoke method of the underlying IDispatch implementation. To specify an Automation dispatch ID for a method, include the dispid directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.

A method declared in a dispatch interface cannot contain directives other than dispid. Parameter and result types must be automatable. In other words, they must be Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool, or any interface type.

Dispatch interface properties

Properties of a dispatch interface do not include access specifiers. They can be declared as read only or write only. To specify a dispatch ID for a property, include the dispid directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as default. No other directives are allowed in dispatch-interface property declarations.

Accessing Automation Objects

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The `CreateOleObject` function (defined in `ComObj`) returns an `IDispatch` reference to an Automation object and is assignment-compatible with the variant `Word`.

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of `varByte` are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the `VarArrayLock` and `VarArrayUnlock` routines.

Automation Object Method-Call Syntax

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both positional and named parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the `:=` symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,, 'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type `Byte`, `Smallint`, `Integer`, `Single`, `Double`, `Currency`, `TDateTime`, `AnsiString`, `WordBool`, or `Variant`. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

Dual Interfaces

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from `IDispatch`.

All methods of a dual interface (except from those inherited from `IInterface` and `IDispatch`) must use the safecall convention, and all method parameter and result types must be automatable. (The automatable types are Byte, Currency, Real, Double, Real48, Integer, Single, Smallint, AnsiString, ShortString, TDateTime, Variant, OleVariant, and WordBool.)

Memory Management

This section describes memory management issues related to programming in Delphi on Win32, and on .NET.

In This Section

[Memory Management on the Win32 Platform](#)

Describes how programs use memory on the Win32 platform.

[Internal Data Formats](#)

Describes the internal data formats of Delphi data types.

[Memory Management Issues on the .NET Platform](#)

Describes memory management issues with Delphi on the .NET platform.

Memory Management on the Win32 Platform

The following material describes how memory management on Win32 is handled, and briefly describes memory issues of variables.

The Memory Manager (Win32 Only)

The memory manager manages all dynamic memory allocations and deallocations in an application. The [New](#), [Dispose](#), [GetMem](#), [ReallocMem](#), and [FreeMem](#) standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

The memory manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of [GlobalAlloc](#), [LocalAlloc](#), and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the memory manager interfaces directly with the Win32 virtual memory API (the [VirtualAlloc](#) and [VirtualFree](#) functions). The memory manager reserves memory from the operating system in 1Mb sections of address space, and commits memory as required in 16K increments. It decommits and releases unused memory in 16K and 1Mb sections. For smaller blocks, committed memory is further suballocated.

Memory manager blocks are always rounded upward to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.

The [System](#) unit provides two procedures, [GetMemoryManager](#) and [SetMemoryManager](#), that allow applications to intercept low-level memory manager calls. The memory manager maintains two status variables, [AllocMemCount](#) and [AllocMemSize](#), which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging. The [System](#) unit also provides a function called [GetHeapStatus](#) that returns a record containing detailed memory-manager status information.

Variables

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

On Win32, an application's stack is defined by two values: the minimum stack size and the maximum stack size. The values are controlled through the [\\$MINSTACKSIZE](#) and [\\$MAXSTACKSIZE](#) compiler directives, and default to 16,384 (16K) and 1,048,576 (1Mb) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Win32 application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an [EStackOverflow](#) exception is raised. (Stack overflow checking is completely automatic. The [\\$S](#) compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

Dynamic variables created with the [GetMem](#) or [New](#) procedure are heap-allocated and persist until they are deallocated with [FreeMem](#) or [Dispose](#).

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

Internal Data Formats

The following topics describe the internal formats of Delphi data types.

Integer Types

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range 128..127 (Shortint), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (Byte), the variable is stored as an unsigned byte.
- If both bounds are within the range 32768..32767 (Smallint), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (Word), the variable is stored as an unsigned word.
- If both bounds are within the range 2147483648..2147483647 (Longint), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (Longword), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (Int64).

Note: a "word" occupies two bytes.

Character Types

On the Win32 platform, Char, an AnsiChar, or a subrange of a Char type is stored as an unsigned byte. A WideChar is stored as an unsigned word.

On the .NET platform, a Char is equivalent to WideChar.

Boolean Types

A Boolean type is stored as a Byte, a ByteBool is stored as a Byte, a WordBool type is stored as a Word, and a LongBool is stored as a Longint.

A Boolean can assume the values 0 (False) and 1 (True). ByteBool, WordBool, and LongBool types can assume the values 0 (False) or nonzero (True).

Enumerated Types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the `{ $Z1 }` state (the default). If an enumerated type has more than 256 values, or if the type was declared in the `{ $Z2 }` state, it is stored as an unsigned word. If an enumerated type is declared in the `{ $Z4 }` state, it is stored as an unsigned double-word.

Real Types

The real types store the binary representation of a sign (+ or -), an exponent, and a significand. A real value has the form

$\pm \text{significand} * 2^{\text{exponent}}$

where the *significand* has a single bit to the left of the binary decimal point. (That is, $0 \leq \text{significand} < 2$.)

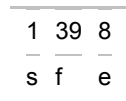
In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses.

For example, for a Real48 value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real48 type

The following discussion of the Real48 type applies only to the Win32 platform. The Real48 type is not supported on the .NET platform.

On the Win32 platform, a 6-byte (48-bit) Real48 number is divided into three fields:



If $0 < e \leq 255$, the value *v* of the number is given by

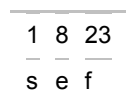
$$v = (1)^s * 2^{(e-129)} * (1.f)$$

If $e = 0$, then $v = 0$.

The Real48 type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a Real48, while NaNs and infinities produce an overflow error if an attempt is made to store them in a Real48.

The Single type

A 4-byte (32-bit) Single number is divided into three fields



The value *v* of the number is given by

$$\text{if } 0 < e < 255, \text{ then } v = (1)^s * 2^{(e-127)} * (1.f)$$

$$\text{if } e = 0 \text{ and } f \neq 0, \text{ then } v = (1)^s * 2^{(126)} * (0.f)$$

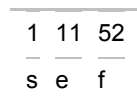
$$\text{if } e = 0 \text{ and } f = 0, \text{ then } v = (1)^s * 0$$

$$\text{if } e = 255 \text{ and } f = 0, \text{ then } v = (1)^s * \text{Inf}$$

$$\text{if } e = 255 \text{ and } f \neq 0, \text{ then } v \text{ is a NaN}$$

The Double type

An 8-byte (64-bit) Double number is divided into three fields



The value *v* of the number is given by

$$\text{if } 0 < e < 2047, \text{ then } v = (1)^s * 2^{(e-1023)} * (1.f)$$

$$\text{if } e = 0 \text{ and } f \neq 0, \text{ then } v = (1)^s * 2^{(1022)} * (0.f)$$

$$\text{if } e = 0 \text{ and } f = 0, \text{ then } v = (1)^s * 0$$

$$\text{if } e = 2047 \text{ and } f = 0, \text{ then } v = (1)^s * \text{Inf}$$

$$\text{if } e = 2047 \text{ and } f \neq 0, \text{ then } v \text{ is a NaN}$$

The Extended type

A 10-byte (80-bit) Extended number is divided into four fields:

1	15	1	63
s	e	i	f

The value v of the number is given by

if $0 \leq e < 32767$, then $v = (1)^s * 2^{(e16383)} * (i.f)$

if $e = 32767$ and $f = 0$, then $v = (1)^s * \text{Inf}$

if $e = 32767$ and $f < 0$, then v is a NaN

Note: On the .NET platform, the Extended type is aliased to Double, and has been deprecated.

The Comp type

An 8-byte (64-bit) Comp number is stored as a signed 64-bit integer.

Note: On the .NET platform, the Comp type is aliased to Int64, and has been deprecated.

The Currency type

An 8-byte (64-bit) Currency number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Pointer Types

A Pointer type is stored in 4 bytes as a 32-bit address. The pointer value nil is stored as zero.

Note: On the .NET platform, the size of a pointer will vary at runtime. Therefore, `sizeof(pointer)` is not a compile-time constant, as it is on the Win32 platform.

Short String Types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (`string[255]`).

Note: On the .NET platform, the short string type is implemented as an array of unsigned bytes.

Long String Types

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is nil and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

Long string dynamic memory layout (Win32 only)

Offset	Contents
-8	32-bit reference-count
-4	length in bytes
0..Length - 1	character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of -1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of -1.

Note: On the .NET platform, an `AnsiString` is implemented as an array of unsigned bytes. The information above on string constants and literals does not apply to the .NET platform.

Wide String Types

On Win32, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is nil and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide string memory block on Windows.

Wide string dynamic memory layout (Win32 only)

Offset	Contents
-4	32-bit length indicator (in bytes)
0..Length - 1	character string
Length	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

On the .NET platform, `String` and `WideString` types are implemented using the `System.String` type. An empty string is not a nil pointer, and the string data is not terminated with a null character.

Set Types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to $(Max \div 8) (Min \div 8) + 1$, where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is $(E \div 8) (Min \div 8)$ and the bit number within that byte is $E \bmod 8$, where *E* denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic Integer type or if the program contains code that takes the address of the set.

Note: On the .NET platform, sets containing more than 32 elements are implemented as an array of bytes. The set type in Delphi for .NET is not CLS compliant, therefore other .NET languages cannot use them.

Static Array Types

On the Win32 platform, a static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

On the .NET platform, static arrays are implemented using the `System.Array` type. Memory layout is therefore determined by the `System.Array` type.

Dynamic Array Types

On the Win32 platform, a dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is nil and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

Dynamic array memory layout (Win32 only)

Offset	Contents
-8	32-bit reference-count
-4	32-bit length indicator (number of elements)
$0..Length * (\text{size of element}) - 1$	array elements

On the .NET platform, dynamic arrays and open array parameters are implemented using the `System.Array` type. As with static arrays, memory layout is therefore determined by the `System.Array` type.

Record Types

On the .NET platform, field layout in record types is determined at runtime, and can vary depending on the architecture of the target hardware. The following discussion of record alignment applies to the Win32 platform only (packed records are supported on the .NET platform, however).

When a record type is declared in the `{ $A+ }` state (the default), and when the declaration does not include a packed modifier, the type is an unpacked record type, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field and by whether fields are declared together. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

Type alignment masks (Win32 only)

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for <i>Real48</i> , 4 for <i>Single</i> , 8 for <i>Double</i> and <i>Extended</i>
Short string types	1
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record

Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	determined by the \$A directive.

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

If two fields share a common type specification, they are packed even if the declaration does not include the packed modifier and the record type is not declared in the `{ $A- }` state. Thus, for example, given the following declaration

```
type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;
```

`A` and `B` are packed (aligned on byte boundaries) because they share the same type specification. The compiler pads the structure with unused bytes to ensure that `C` appears on a quadword boundary.

When a record type is declared in the `{ $A- }` state, or when the declaration includes the packed modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

File Types

The following discussion of file types applies to the Win32 platform only. On the .NET platform, text files are implemented with a class (as opposed to a record). Binary file types (e.g. `File of MyType`) are not supported on the .NET platform.

On the Win32 platform, file types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```
type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of Char; );
    end;
```

Text files occupy 460 bytes, which are laid out as follows:

```

type
    TTextBuf = array[0..127] of Char;
    TTextRec = packed record
        Handle: Integer;
        Mode: word;
        Flags: word;
        BufSize: Cardinal;
        BufPos: Cardinal;
        BufEnd: Cardinal;
        BufPtr: PChar;
        OpenFunc: Pointer;
        InOutFunc: Pointer;
        FlushFunc: Pointer;
        CloseFunc: Pointer;
        UserData: array[1..32] of Byte;
        Name: array[0..259] of Char;
        Buffer: TTextBuf;
    end;

```

Handle contains the file's handle (when the file is open).

The `Mode` field can assume one of the values

```

const
    fmClosed = $D7B0;
    fmInput = $D7B1;
    fmOutput = $D7B2;
    fmInOut = $D7B3;

```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file; see Device functions. Flags determines the line break style as follows:

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

All other Flags bits are reserved for future use.

Procedural Types

On the Win32 platform, a procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

On the .NET platform, procedural types are implemented using the `System.MulticastDelegate` class types.

Class Types

The following discussion of the internal layout of class types applies to the Win32 platform only. On the .NET platform, class layout is determined at runtime. Runtime type information is obtained using the System.Reflection APIs in the .NET framework.

On the Win32 platform, a class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object's fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMT's are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMT's, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointers one per user-defined virtual method in the class type in order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Delphi's implementation. Applications should use the methods defined in TObject to query this information, since the layout is likely to change in future implementations of the Delphi language.

Virtual method table layout (Win32 Only)

Offset	Type	Description
-76	Pointer	pointer to virtual method table (or nil)
-72	Pointer	pointer to interface table (or nil)
-68	Pointer	pointer to Automation information table (or nil)
-64	Pointer	pointer to instance initialization table (or nil)
-60	Pointer	pointer to type information table (or nil)
-56	Pointer	pointer to field definition table (or nil)
-52	Pointer	pointer to method definition table (or nil)
-48	Pointer	pointer to dynamic method table (or nil)
-44	Pointer	pointer to short string containing class name
-40	Cardinal	instance size in bytes
-36	Pointer	pointer to a pointer to ancestor class (or nil)
-32	Pointer	pointer to entry point of <i>SafecallException</i> method (or nil)
-28	Pointer	entry point of <i>AfterConstruction</i> method
-24	Pointer	entry point of <i>BeforeDestruction</i> method
-20	Pointer	entry point of <i>Dispatch</i> method
-16	Pointer	entry point of <i>DefaultHandler</i> method
-12	Pointer	entry point of <i>NewInstance</i> method
-8	Pointer	entry point of <i>FreeInstance</i> method
-4	Pointer	entry point of <i>Destroy</i> destructor
0	Pointer	entry point of first user-defined virtual method
4	Pointer	entry point of second user-defined virtual method

Class Reference Types

On the Win32 platform, a class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

On the .NET platform, class reference types are implemented using compiler-constructed nested classes inside the class type they support. These implementation details are subject to change in future compiler releases.

Variant Types

The following discussion of the internal layout of variant types applies to the Win32 platform only. On the .NET platform, variants are an alias of `System.Object`. Variants rely on boxing and unboxing of data into an object wrapper, as well as Delphi helper classes to implement the variant-related RTL functions.

On the Win32 platform, a variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The `System` and `Variants` units define constants and types for variants.

The `TVarData` type represents the internal structure of a Variant variable (on Windows, this is identical to the Variant type used by COM and the Win32 API). The `TVarData` type can be used in typecasts of Variant variables to access the internal structure of a variable. The `TVarData` record contains the following fields:

- `VType` contains the type code of the variant in the lower twelve bits (the bits defined by the `varTypeMask` constant). In addition, the `varArray` bit may be set to indicate that the variant is an array, and the `varByRef` bit may be set to indicate that the variant contains a reference as opposed to a value.
- The `Reserved1`, `Reserved2`, and `Reserved3` fields are unused.

The contents of the remaining eight bytes of a `TVarData` record depend on the `VType` field as follows:

- If neither the `varArray` nor the `varByRef` bits are set, the variant contains a value of the given type.
- If the `varArray` bit is set, the variant contains a pointer to a `TVarArray` structure that defines an array. The type of each array element is given by the `varTypeMask` bits in the `VType` field.
- If the `varByRef` bit is set, the variant contains a reference to a value of the type given by the `varTypeMask` and `varArray` bits in the `VType` field.

The `varString` type code is private. Variants containing a `varString` value should never be passed to a non-Delphi function. On Win32, Delphi's Automation support automatically converts `varString` variants to `varOleStr` variants before passing them as parameters to external functions.

Memory Management Issues on the .NET Platform

The .NET Common Language Runtime is a garbage-collected environment. This means the programmer is freed (for the most part) from worrying about memory allocation and deallocation. Broadly speaking, after you allocate memory, the CLR determines when it is safe to free that memory. "Safe to free" means that no more references to that memory exist.

This topic covers the following memory management issues:

- Creating and destroying objects
- Unit initialization and finalization sections
- Unit initialization and finalization in assemblies and packages

Constructors

In Delphi for .NET, a constructor must always call an inherited constructor before it may access or initialize any inherited class members. The compiler generates an error if your constructor code does not call the inherited constructor (a valid situation in Delphi for Win32), but it is important to examine your constructors to make sure that you do not access any inherited class fields, directly or indirectly, before the call to the inherited constructor.

Note: A constructor can initialize fields from its own class, prior to calling the inherited constructor.

Finalization

Every class in the .NET Framework (including VCL.NET classes) inherits a method called `Finalize`. The garbage collector calls the `Finalize` method when the memory for the object is about to be freed. Since the method is called by the garbage collector, you have no control over when it is called. The asynchronous nature of finalization is a problem for objects that open resources such as file handles and database connections, because the `Finalize` method might not be called for some time, leaving these connections open.

To add a finalizer to a class, override the `strict protected` `Finalize` procedure that is inherited from `TObject`. The .NET platform places limits on what you can do in a finalizer, because it is called when the garbage collector is cleaning up objects. The finalizer may execute in a different thread than the thread the object was created in. A finalizer cannot allocate new memory, and cannot make calls outside of itself. If your class has references to other objects, a finalizer can refer to them (that is, their memory is guaranteed not to have been freed yet), but be aware that their state is undefined, as you do not know whether they have been finalized yet.

When a class has a finalizer, the CLR must add newly instantiated objects of the class to the finalization list. Further, objects with finalizers tend to persist in memory longer, as they are not freed when the garbage collector first determines that they are no longer actively referenced. If the object has references to other objects, those objects are also not freed right away (even if they don't have finalizers themselves), but must also persist in memory until the original object is finalized. Therefore, finalizers do impart a fair amount of overhead in terms of memory consumption and execution performance, so they should be used judiciously.

It is a good practice to restrict finalizers to small objects that represent unmanaged resources. Classes that use these resources can then hold a reference to the small object with the finalizer. In this way, big classes, and classes that reference many other classes, do not hoard memory because of a finalizer.

Another good practice is to suppress finalizers when a particular resource has already been released in a destructor. After freeing the resources, you can call `SuppressFinalize`, which causes the CLR to remove the object from the finalization list. Be careful not to call `SuppressFinalize` with a nil reference, as that causes a runtime exception.

The Dispose Pattern

Another way to free up resources is to implement the *dispose* pattern. Classes adhering to the dispose pattern must implement the .NET interface called `IDisposable`. `IDisposable` contains only one method, called `Dispose`. Unlike the `Finalize` method, the `Dispose` method is public. It can be called directly by a user of the class, as opposed to relying on the garbage collector to call it. This gives you back control of freeing resources, but calling `Dispose` still does not reclaim memory for the object itself - that is still for the garbage collector to do. Note that some classes in the .NET Framework implement both `Dispose`, and another method such as `Close`. Typically the `Close` method simply calls `Dispose`, but the extra method is provided because it seems more "natural" for certain classes such as files.

Delphi for .NET classes are free to use the `Finalize` method for freeing system resources, however the recommended method is to implement the dispose pattern. The Delphi for .NET compiler recognizes a very specific destructor pattern in your class, and implements the `IDisposable` interface for you. This enables you to continue writing new code for the .NET platform the same way you always have, while allowing much of your existing Win32 Delphi code to run in the garbage collected environment of the CLR.

The compiler recognizes the following specific pattern of a Delphi destructor:

```
TMyClass = class(TObject)
    destructor Destroy; override;
end;
```

Your destructor must fit this pattern exactly:

- The name of the destructor must be `Destroy`.
- The keyword `override` must be specified.
- The destructor cannot take any parameters.

In the compiler's implementation of the dispose pattern, the `Free` method is written so that if the class implements the `IDisposable` interface (which it does), then the `Dispose` method is called, which in turn calls your destructor.

You can still implement the `IDisposable` interface directly, if you choose. However, the compiler's automatic implementation of the `Free-Dispose-Destroy` mechanism cannot coexist with your implementation of `IDisposable`. The two methods of implementing `IDisposable` are mutually exclusive. You must choose to either implement `IDisposable` directly, or equip your class with the familiar `destructor Destroy; override` pattern and rely on the compiler to do the rest. The `Free` method will call `Dispose` in either case, but if you implement `Dispose` yourself, you must call your destructor yourself. If you want to implement `IDisposable` yourself, your destructor cannot be called `Destroy`.

Note: You can declare destructors with other names; the compiler only provides the `IDisposable` implementation when the destructor fits the above pattern.

The `Dispose` method is not called automatically; the `Free` method must be called in order for `Dispose` to be called. If an object is freed by the garbage collector because there are no references to it, but you did not explicitly call `Free` on the object, the object will be freed, but the destructor will not execute.

Note: When the garbage collector frees the memory used by an object, it also reclaims the memory used by all fields of the object instance as well. This means the most common reason for implementing destructors in Delphi for Win32 - to release allocated memory - no longer applies. However, in most cases, unmanaged resources such as window handles or file handles still need to be released.

To eliminate the possibility of destructors being called more than once, the Delphi for .NET compiler introduces a field called `DisposeCount` into every class declaration. If the class already has a field by this name, the name collision will cause the compiler to produce a syntax error in the destructor.

Unit Initialization and Finalization

On the .NET platform, units that you depend on will be initialized prior to initializing your own unit. However, there is no way to guarantee the order in which units are initialized. Nor is there a way to guarantee when they will be initialized. Be aware of initialization code that depends on another unit's initialization side effects, such as the creation of a file. Such a dependency cannot be made to work reliably on the .NET platform.

Unit finalization is subject to the same constraints and difficulties as the `Finalize` method of objects. Specifically, unit finalization is asynchronous, and, there no way to determine when it will happen (or if it will happen, though under most circumstances, it will).

Typical tasks performed in a unit finalization include freeing global objects, unregistering objects that are used by other units, and freeing resources. Because .NET is a memory managed environment, the garbage collector will free global objects even if the unit finalization section is not called. The units in an application domain are loaded and unloaded together, so you do not need to worry about unregistering objects. All units that can possibly refer to each other (even in different assemblies) are released at the same time. Since object references do not cross application domains, there is no danger of something keeping a dangling reference to an object type or code that has been unloaded from memory.

Freeing resources (such as file handles or window handles) is the most important consideration in unit finalization. Because unit finalization sections are not guaranteed to be called, you may want to rewrite your code to handle this issue using finalizers rather than relying on the unit finalization.

The main points to keep in mind for unit initialization and finalization on the .NET platform are:

- 1 The `Finalize` method is called asynchronously (both for objects, and for units).
- 2 Finalization and destructors are used to free unmanaged resources such as file handles. You do not need to destroy object member variables; the garbage collector takes care of this for you.
- 3 Classes should rely on the compiler's implementation of `IDisposable`, and provide a destructor called `Destroy`.
- 4 If a class implements `IDisposable` itself, it cannot have a destructor called `Destroy`.
- 5 Reference counting is deprecated. Try to use the `destructor Destroy; override;` pattern wherever possible.
- 6 Unit initialization should not depend on side effects produced by initialization of dependent units.

Unit Initialization Considerations for Assemblies and Dynamically Linked Packages

Under Win32, the Delphi compiler uses the `DllMain` function as a hook from which to execute unit initialization code. No such execution path exists in the .NET environment. Fortunately, other means of implementing unit initialization exist in the .NET Framework. However, the differences in the implementation between Win32 and .NET could impact the order of unit initialization in your application.

The Delphi for .NET compiler uses CLS-compliant class constructors to implement unit initialization hooks. The CLR requires that every object type have a class constructor. These constructors, or type initializers, are guaranteed to be executed *at most* one time. Class constructors are executed at most one time, because in order for the type to be loaded, it must be used. That is, the assembly containing a type will not be loaded until the type is actually used at runtime. If the assembly is never loaded, its unit initialization section will never run.

Circular unit references also impact the unit initialization process. If unit A uses unit B, and unit B then uses unit A in its implementation section, the order of unit initialization is undefined. To fully understand the possibilities, it is helpful to look at the process one step at a time.

- 1 Unit A's initialization section uses a type from unit B. If this is the first reference to the type, the CLR will load its assembly, triggering the unit initialization of unit B.
- 2 As a consequence, loading and initializing unit B occurs before unit A's initialization section has completed execution. Note this is a change from how unit initialization works under Win32.

- 3 Suppose that unit B's initialization is in progress, and that a type from unit A is used. Unit A has not completed initialization, and such a reference could cause an access violation.

The unit initialization should only use types defined within that unit. Using types from outside the unit will impact unit initialization, and could cause an access violation, as noted above.

Unit initialization for DLLs happens automatically; it is triggered when a type within the DLL is referenced. Applications created with other .NET languages can use Delphi for .NET assemblies without concern for the details of unit initialization.

Program Control

This section describes how parameters are passed to procedures and functions.

In This Section

[Program Control](#)

Describes parameters, functions, method calls, and exit procedures.

Program Control

The concepts of passing parameters and function result processing are important to understand before you undertake your application projects. Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

The following topics are covered in this material:

- Passing Parameters.
- Handling Function Results.
- Handling Method Calls.
- Understanding Exit Procedures.

Passing Parameters

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see the topic on Calling Conventions.

By Value vs. By Reference

Variable (var) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (const) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A Single parameter occupies 4 bytes, and a Double, Comp, or Currency parameter occupies 8 bytes. A Real48 occupies 8 bytes, with the Real48 value stored in the lower 6 bytes. An Extended occupies 12 bytes, with the Extended value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.
- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value `nil` is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the register and pascal conventions, a variant parameter is passed as a 32-bit pointer to a Variant value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32-bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack under the cdecl, stdcall, and safecall conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.
- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

Pascal, cdecl, stdcall, and safecall Conventions

Under the pascal, cdecl, stdcall and safecall conventions, all parameters are passed on the stack. Under the pascal convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up

at the highest address and the last parameter ends up at the lowest address. Under the `cdecl`, `stdcall`, and `safecall` conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

Register Convention

Under the register convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the pascal convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, Int64, and structured types do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to `Test` passes A in EAX as a 32-bit integer, B in EDX as a pointer to a Char, and D in ECX as a pointer to a long-string memory block; C and E are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a `CLD` instruction) and must return with the direction flag cleared.

Note: Delphi language procedures and functions are generally invoked with the assumption that the FPU stack is empty: The compiler tries to use all eight FPU stack entries when it generates code.

When working with the MMX and XMM instructions, be sure to preserve the values of the xmm and mm registers. Delphi functions are invoked with the assumption that the x87 FPU data registers are available for use by x87 floating point instructions. That is, the compiler assumes that the EMMS/FEMMS instruction has been called after MMX operations. Delphi functions do not make any assumptions about the state and content of xmm registers. They do not guarantee that the content of xmm registers is unchanged.

Handling Function Results

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type `Currency`, the value in ST(0) is scaled by 10000. For example, the `Currency` value 1.234 is returned in ST(0) as 12340.
- For a string, dynamic array, method pointer, or variant result, the effects are the same as if the function result were declared as an additional `var` parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Int64 is returned in EDX:EAX.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.
- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional `var` parameter that is passed to the function after the declared parameters.

Handling Method Calls

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter `Self`, which is a reference to the instance or class in which the method is called. The `Self` parameter is passed as a 32-bit pointer.

- Under the register convention, `Self` behaves as if it were declared before all other parameters. It is therefore always passed in the `EAX` register.
- Under the pascal convention, `Self` behaves as if it were declared after all other parameters (including the additional var parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.
- Under the `cdecl`, `stdcall`, and `safecall` conventions, `Self` behaves as if it were declared before all other parameters, but after the additional var parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional var parameter.

Constructors and destructors use the same calling conventions as other methods, except that an additional Boolean flag parameter is passed to indicate the context of the constructor or destructor call.

A value of `False` in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the `inherited` keyword. In this case, the constructor behaves like an ordinary method. A value of `True` in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by `Self`, and returns a reference to the newly created object in `EAX`.

A value of `False` in the flag parameter of a destructor call indicates that the destructor was invoked using the `inherited` keyword. In this case, the destructor behaves like an ordinary method. A value of `True` in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by `Self` just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the register convention, it is passed in the `DL` register. Under the pascal convention, it is pushed before all other parameters. Under the `cdecl`, `stdcall`, and `safecall` conventions, it is pushed just before the `Self` parameter.

Since the `DL` register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of `DL` before exiting so that `BeforeDestruction` or `AfterConstruction` can be called properly.

Understanding Exit Procedures

Exit procedures ensure that specific actions such as updating and closing files are carried out before a program terminates. The `ExitProc` pointer variable allows you to *install* an exit procedure, so that it is always called as part of the program's termination whether the termination is normal, forced by a call to `Halt`, or the result of a runtime error. An exit procedure takes no parameters.

Note: It is recommended that you use finalization sections rather than exit procedures for all exit behavior. `Exit` procedures are available only for executables. For `.DLLs` (Win32) you can use a similar variable, `DllProc`, which is called when the library is loaded as well as when it is unloaded. For packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn't executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents of `ExitProc` before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of `ExitProc`.

The following code shows a skeleton implementation of an exit procedure.

```
var
ExitSave: Pointer;

procedure MyExit;

begin
    ExitProc := ExitSave; // always restore old vector first
    .
    .
end;

begin
    ExitSave := ExitProc;
    ExitProc := @MyExit;
    .
    .
end.
```

On entry, the code saves the contents of `ExitProc` in `ExitSave`, then installs the `MyExit` procedure. When called as part of the termination process, the first thing `MyExit` does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until `ExitProc` becomes `nil`. To avoid infinite loops, `ExitProc` is set to `nil` before every call, so the next exit procedure is called only if the current exit procedure assigns an address to `ExitProc`. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the `ExitCode` integer variable and the `ErrorAddr` pointer variable. In case of normal termination, `ExitCode` is zero and `ErrorAddr` is `nil`. In case of termination through a call to `Halt`, `ExitCode` contains the value passed to `Halt` and `ErrorAddr` is `nil`. In case of termination due to a runtime error, `ExitCode` contains the error code and `ErrorAddr` contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the Input and Output files. If `ErrorAddr` is not `nil`, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines `ErrorAddr` and outputs a message if it's not `nil`; before returning, set `ErrorAddr` to `nil` so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in `ExitCode` as a return code.

Inline Assembly Code (Win32 Only)

This section describes the use of the inline assembler on the Win32 platform.

In This Section

[Using Inline Assembly Code \(Win32 Only\)](#)

Describes the inline assembler, which is available in the Win32 Delphi compiler only.

[Assembly Expressions \(Win32 Only\)](#)

Describes assembly language expressions and how they are used.

[Assembly Procedures and Functions \(Win32 Only\)](#)

Describes the use of standalone assembly language procedures and functions.

Using Inline Assembly Code (Win32 Only)

The built-in assembler allows you to write assembly code within Delphi programs. The inline assembler is available only on the Win32 Delphi compiler. It has the following features:

- Allows for inline assembly.
- Supports all instructions found in the Intel Pentium 4, Intel MMX extensions, Streaming SIMD Extensions (SSE), and the AMD Athlon (including 3D Now!).
- Provides no macro support, but allows for pure assembly function procedures.
- Permits the use of Delphi identifiers, such as constants, types, and variables in assembly statements.

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See the topic on External declarations for more information. If you have external assembly code that you want to use in your applications, you should consider rewriting it in the Delphi language or minimally reimplement it using the inline assembler.

Using the asm Statement

The built-in assembler is accessed through asm statements, which have the form

```
asm statementList end
```

where *statementList* is a sequence of assembly statements separated by semicolons, end-of-line characters, or Delphi comments.

Comments in an asm statement must be in Delphi style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word `inline` and the directive `assembler` are maintained for backward compatibility only. They have no effect on the compiler.

Using Registers

In general, the rules of register use in an asm statement are the same as those of an external procedure or function. An asm statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an asm statement, EBP points to the current stack frame and ESP points to the top of the stack. Except for ESP and EBP, an asm statement can assume nothing about register contents on entry to the statement.

Understanding Assembler Syntax (Win32 Only)

The inline assembler is available only on the Win32 Delphi compiler. The following material describes the elements of the assembler syntax necessary for proper use.

- Assembler Statement Syntax
- Labels
- Instruction Opcodes
- Assembly Directives
- Operands

Assembler Statement Syntax

This syntax of an assembly statement is

```
Label: Prefix Opcode Operand1, Operand2
```

where *Label* is a label, *Prefix* is an assembly prefix opcode (operation code), *Opcode* is an assembly instruction opcode or directive, and *Operand* is an assembly expression. Label and Prefix are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembly statements, but not within them. For example,

```
MOV AX,1 {Initial value}           { OK }
MOV CX,100 {Count}                  { OK }

      MOV {Initial value} AX,1;     { Error! }
MOV CX, {Count} 100                 { Error! }
```

Labels

Labels are used in built-in assembly statements as they are in the Delphi language by writing the label and a colon before a statement. There is no limit to a label's length. As in Delphi, labels must be declared in a label declaration part in the block containing the asm statement. The one exception to this rule is local labels.

Local labels are labels that start with an at-sign (@). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to asm statements, and the scope of a local label extends from the asm reserved word to the end of the asm statement that contains it. A local label doesn't have to be declared.

Instruction Opcodes

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family
- Pentium Pro and Pentium II
- Pentium III
- Pentium 4

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow! (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

RET instruction sizing

The RET instruction opcode always generates a near return.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is 128 to 127 bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is 128 to 127 bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembly statement

```
JC      Stop
```

where Stop isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
JNC     Skip
JMP     Stop
Skip:
```

Jumps to the entry points of procedures and functions are always near.

Assembly Directives

The built-in assembler supports three assembly define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

Directive	Description
DB	Define byte: generates a sequence of bytes. Each operand can be a constant expression with a value between 128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.
DW	Define word: generates a sequence of words. Each operand can be a constant expression with a value between 32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer, a word that contains the offset part of the address.
DD	Define double word: generates a sequence of double words. Each operand can be a constant expression with a value between 2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

DQ Define quad word: defines a quad word for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembly statements. To generate uninitialized or initialized data in the data segment, you should use Delphi var or const declarations.

Some examples of DB, DW, and DD directives follow.

```
asm
  DB
FFH                                     { One byte }
  DB      0,99
{ Two bytes }
  DB      'A'
{ Ord('A') }
  DB      'Hello world...',0DH,0AH      { String followed by CR/LF }
  DB      12,'string'                    { Delphi style string }
  DW      0FFFFH                          { One word }
  DW      0,9999
{ Two words }
  DW      'A'
{ Same as DB 'A',0 }
  DW      'BA'
{ Same as DB 'A','B' }
  DW      MyVar
{ Offset of MyVar }
  DW      MyProc
{ Offset of MyProc }
  DD      0FFFFFFFFH                      { One double-word }
  DD      0,999999999                    { Two double-words }
  DD      'A'
{ Same as DB 'A',0,0,0 }
  DD      'DCBA'
{ Same as DB 'A','B','C','D' }
  DD      MyVar
{ Pointer to MyVar }
  DD      MyProc
{ Pointer to MyProc }
end;
```

When an identifier precedes a DB, DW, or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, the assembler allows the following:

```
ByteVar      DB      ?
WordVar      DW      ?
IntVar       DD      ?

.
.
.
MOV          AL,ByteVar
MOV          BX,WordVar
MOV          ECX,IntVar
```

The built-in assembler doesn't support such variable declarations. The only kind of symbol that can be defined in an inline assembly statement is a label. All variables must be declared using Delphi syntax; the preceding construction can be replaced by

```

var
    ByteVar: Byte;
    WordVar: Word;
    IntVar: Integer;
    .
    .
    .

asm
    MOV AL,ByteVar
    MOV BX,WordVar
    MOV ECX,IntVar
end;

```

SMALL and LARGE can be used to determine the width of a displacement:

```
MOV EAX, [LARGE $1234]
```

This instruction generates a 'normal' move with a 32-bit displacement (\$00001234).

```
MOV EAX, [SMALL $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement (\$1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
MOV EAX, [SMALL 123]
```

as opposed to

```
MOV EAX, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual methods: VMTOFFSET and DMTINDEX.

VMTOFFSET retrieves the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter (for example, TExample.VirtualMethod), or an interface name and an interface method name.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example, TExample.DynamicMethod. To invoke the dynamic method, call System.@CallDynInst with the (E)SI register containing the value obtained from DMTINDEX.

Note: Methods with the *message* directive are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```

TMyClass = class
    procedure x; message MYMESSAGE;
end;

```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```

program Project2;
type
  TExample = class
    procedure DynamicMethod; dynamic;
    procedure VirtualMethod; virtual;
  end;

  procedure TExample.DynamicMethod;
begin
  end;

  procedure TExample.VirtualMethod;
begin
  end;

  procedure CallDynamicMethod(e: TExample);
asm
  // Save ESI register
  PUSH    ESI

  // Instance pointer needs to be in EAX
  MOV     EAX, e

  // DMT entry index needs to be in (E)SI
  MOV     ESI, DMTINDEX TExample.DynamicMethod

  // Now call the method
  CALL    System.@CallDynaInst

  // Restore ESI register
  POP     ESI

end;

  procedure CallVirtualMethod(e: TExample);
asm
  // Instance pointer needs to be in EAX
  MOV     EAX, e

  // Retrieve VMT table entry
  MOV     EDX, [EAX]

  // Now call the method at offset VMTOFFSET
  CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]

end;

  var
    e: TExample;
begin
  e := TExample.Create;
  try
    CallDynamicMethod(e);
    CallVirtualMethod(e);
  finally
    e.Free;
  end;
end;

```

```
end.
```

Operands

Inline assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings:

Built-in assembler reserved words

AH	CL	DX	ESP	mm4	SHL	WORD
AL	CS	EAX	FS	mm5	SHR	xmm0
AND	CX	EBP	GS	mm6	SI	xmm1
AX	DH	EBX	HIGH	mm7	SMALL	xmm2
BH	DI	ECX	LARGE	MOD	SP	xmm3
BL	DL	EDI	LOW	NOT	SS	xmm4
BP	CL	EDX	mm0	OFFSET	ST	xmm5
BX	DMTINDEX	EIP	mm1	OR	TBYTE	xmm6
BYTE	DS	ES	mm2	PTR	TYPE	xmm7
CH	DWORD	ESI	mm3	QWORD	VMTOFFSET	XOR

Reserved words always take precedence over user-defined identifiers. For example,

```
var
    Ch: Char;
    .
    .
    .
asm
    MOV     CH, 1
end;
```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) override operator:

```
MOV&Ch, 1
```

It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

Assembly Expressions (Win32 Only)

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants. The inline assembler is available only on the Win32 Delphi compiler.

Expressions are built from expression elements and operators, and each expression has an associated expression class and expression type. This topic covers the following material:

- Differences between Delphi and Assembler Expressions
- Expression Elements
- Expression Classes
- Expression Types
- Expression Operators

Differences between Delphi and Assembler Expressions

The most important difference between Delphi expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value. In other words, it must resolve to a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid statement.

```
asm
  MOV     Z,X+Y
end;
```

Because both `X` and `Y` are constants, the expression `X + Y` is a convenient way of writing the constant 30, and the resulting instruction simply moves the value 30 into the variable `Z`. But if `X` and `Y` are variables

```
var
  X, Y: Integer;
```

the built-in assembler cannot compute the value of `X + Y` at compile time. In this case, to move the sum of `X` and `Y` into `Z` you would use

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

In a Delphi expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Delphi the expression `X + 4` (where `X` is a variable)

means the contents of `X` plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of `X`. So, even though you are allowed to write

```
asm
    MOV     EAX, X+4
end;
```

this code doesn't load the value of `X` plus 4 into `AX`; instead, it loads the value of a word stored four bytes beyond `X`. The correct way to add 4 to the contents of `X` is

```
asm
    MOV     EAX, X
    ADD     EAX, 4
end;
```

Expression Elements

The elements of an expression are constants, registers, and symbols.

Numeric Constants

Numeric constants must be integers, and their values must be between 2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a `B` after the number, octal notation by writing an `O` after the number, and hexadecimal notation by writing an `H` after the number or a `$` before the number.

Numeric constants must start with one of the digits 0 through 9 or the `$` character. When you write a hexadecimal constant using the `H` suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example, `0BAD4H` and `$BAD4` are hexadecimal constants, but `BAD4H` is an identifier because it starts with a letter.

String Constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
'Linux'
"That's all folks"
'"That"'s    all folks," he said.'
'100'
'''
""
```

String constants of any length are allowed in `DB` directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where **Ch1** is the rightmost (last) character and **Ch4** is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

String examples and their values

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers in the inline assembler:

CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registers for example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(X), where X is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Delphi identifiers in assembly language expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol **@Result**, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
    Result := X + Y;
end;
```

could be written in assembly language as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
    asm
        MOV     EAX, X
        ADD     EAX, Y
        MOV     @Result, EAX
    end;
end;
```

The following symbols cannot be used in asm statements:

- Standard procedures and functions (for example, WriteLn and Chr).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The @Result symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in asm statements.

Symbols recognized by the built-in assembler

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable or address of a pointer to the variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
@Result	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV     EAX, Count
```

assembles into `MOV EAX, [EBP4]`.

The built-in assembler treats var parameters as a 32-bit pointers, and the size of a var parameter is always 4. The syntax for accessing a var parameter is different from that for accessing a value parameter. To access the contents of a var parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
    asm
        MOV     EAX, X
        MOV     EAX, [EAX]
        MOV     EDX, Y
        ADD     EAX, [EDX]
        MOV     @Result, EAX
    end;
end;
```

Identifiers can be qualified within asm statements. For example, given the declarations

```
type
    TPoint = record
        X, Y: Integer;
    end;
    TRect = record
        A, B: TPoint;
    end;
var
    P: TPoint;
    R: TRect;
```

the following constructions can be used in an asm statement to access fields.

```
MOV     EAX, P.X
MOV     EDX, P.Y
MOV     ECX, R.A.X
MOV     EBX, R.B.Y
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of [EDX] into EAX.

```
MOV     EAX, (TRect PTR [EDX]).B.X
MOV     EAX, TRect([EDX]).B.X
MOV     EAX, TRect[EDX].B.X
MOV     EAX, [EDX].TRect.B.X
```

Expression Classes

The built-in assembler divides expressions into three classes: registers, memory references, and immediate values.

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Delphi's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Delphi's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;
  .
  .
  .
asm
  MOV      EAX, Start                { MOV EAX,xxxx }
  MOV      EBX, Count                { MOV EBX,[xxxx] }
  MOV      ECX, [Start]              { MOV ECX,[xxxx] }
  MOV      EDX, OFFSET Count         { MOV EDX,xxxx }
end;
```

Because `Start` is an immediate value, the first `MOV` is assembled into a move immediate instruction. The second `MOV`, however, is translated into a move memory instruction, as `Count` is a memory reference. In the third `MOV`, the brackets convert `Start` into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth `MOV`, the `OFFSET` operator converts `Count` into an immediate value (the offset of `Count` in the data segment).

The brackets and `OFFSET` operator complement each other. The following asm statement produces identical machine code to the first two lines of the previous asm statement.

```
asm
  MOV      EAX, OFFSET [Start]
  MOV      EBX, [OFFSET Count]
end;
```

Memory references and immediate values are further classified as either relocatable or absolute. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression Types

Every built-in assembler expression has a type, or more correctly a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an `Integer` variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  .
  .
  .
asm
```

```
MOV     AL,QuitFlag
MOV     BX,OutBufPtr
end;
```

the assembler checks that the size of `QuitFlag` is one (a byte), and that the size of `OutBufPtr` is two (a word). The instruction

```
MOV     DL,OutBufPtr
```

produces an error because `DL` is a byte-sized register and `OutBufPtr` is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

These `MOV` instructions all refer to the first (least significant) byte of the `OutBufPtr` variable.

In some cases, a memory reference is untyped. One example is an immediate value (`Buffer`) enclosed in square brackets:

```
procedure Example(var Buffer);
asm
    MOV AL,    [Buffer]
    MOV CX,    [Buffer]
    MOV EDX,   [Buffer]
end;
```

The built-in assembler permits these instructions, because the expression `[Buffer]` has no type. `[Buffer]` means "the contents of the location indicated by `Buffer`," and the type can be determined from the first operand (byte for `AL`, word for `CX`, and double-word for `EDX`).

In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast. For example,

```
INC     BYTE PTR [ECX]
IMUL    WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Delphi types.

Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Expression Operators

The built-in assembler provides a variety of operators. Precedence rules are different from that of the Delphi language; for example, in an asm statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

Precedence of built-in assembler expression operators

Operators	Remarks	Precedence
&		highest
(...), [...],,, HIGH, LOW		
+, -	unary + and -	
:		
OFFSET, TYPE, PTR, *, /, MOD, SHL, SHR, +, -	binary + and -	
NOT, AND, OR, XOR		lowest

The following table defines the built-in assembler's expression operators.

Definitions of built-in assembler expression operators

Operator	Description
&	Identifier override. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	Subexpression. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[...]	Memory reference. The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	Structure member selector. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	Unary plus. Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
-	Unary minus. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	Addition. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
-	Subtraction. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	Segment override. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.

TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	Typecast operator. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	Multiplication. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	Integer division. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	Remainder after integer division. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	Logical shift left. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	Logical shift right. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	Bitwise negation. The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	Bitwise AND. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	Bitwise OR. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	Bitwise exclusive OR. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembly Procedures and Functions (Win32 Only)

You can write complete procedures and functions using inline assembly language code, without including a `begin...end` statement. This topic covers these issues:

- Compiler Optimizations.
- Function Results.

The inline assembler is available only on the Win32 Delphi compiler.

Compiler Optimizations

An example of the type of function you can write is as follows:

```
function LongMul(X, Y: Integer): Longint;  
asm  
    MOV     EAX,X  
    IMUL Y  
  
end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were var parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a reference to the `@Result` symbol is an error. For strings, variants, and interfaces, the caller always allocates an `@Result` pointer.
- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- Locals is the size of the local variables and Params is the size of the parameters. If both Locals and Params are zero, there is no entry code, and the exit code consists simply of a RET instruction.

The automatically generated entry and exit code for the routine looks like this:

```
PUSH     EBP                                ;Present if Locals <> 0 or Params <> 0  
MOV      EBP,ESP                            ;Present if Locals <> 0 or Params <> 0  
SUB      ESP,Locals                        ;Present if Locals <> 0  
.  
.  
.  
MOV      ESP,EBP                            ;Present if Locals <> 0  
POP      EBP                                ;Present if Locals <> 0 or Params <> 0  
RET      Params                             ;Always present
```

If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.

Function Results

Assembly language functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (Currency values are scaled by 10000.)

- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by `@Result`.

Using .NET Custom Attributes

.NET framework assemblies are self-describing entities. They contain intermediate code that is compiled to native machine instructions when the assembly is loaded. More than that, assemblies contain a wealth of information about that code. The compiler emits this descriptive information, or metadata, into the assembly as it processes the source code. In other programming environments, there is no way to access metadata once your code is compiled; the information is lost during the compilation process. On the .NET platform, however, you have the ability to access metadata using runtime reflection services.

The .NET framework gives you the ability to extend the metadata emitted by the compiler with your own descriptive attributes. These customized attributes are somewhat analogous to language keywords, and are stored with the other metadata in the assembly.

- Declaring custom attributes
- Using custom attributes
- Custom attributes and interfaces

Declaring a Custom Attribute Class

Creating a custom attribute is the same as declaring a class. The custom attribute class has a constructor, and properties to set and retrieve its state data. Custom attributes must inherit from `TCustomAttribute`. The following code declares a custom attribute with a constructor and two properties:

```
type
  TCustomCodeAttribute = class(TCustomAttribute)
  private
    Fprop1 : integer;
    Fprop2 : integer;
    aVal   : integer;
    procedure Setprop1(p1 : integer);
    procedure Setprop2(p2 : integer);
  public
    constructor Create(const myVal : integer);
    property prop1 : integer read Fprop1 write Setprop1;
    property prop2 : integer read Fprop2 write Setprop2;
  end;
```

The implementation of the constructor might look like

```
constructor TCustomCodeAttribute.Create(const myVal : integer);
begin
  inherited Create;
  aVal := myVal;
end;
```

Delphi for .NET supports the creation of custom attribute classes, as shown above, and all of the custom attributes provided by the .NET framework.

Using Custom Attributes

Custom attributes are placed directly before the source code symbol to which the attribute applies. Attributes can be placed before

- variables and constants
- procedures and functions
- function results
- procedure and function parameters
- types
- fields, properties, and methods

Note that Delphi for .NET supports the use of named properties in the initialization. These can be the names of properties, or of public fields of the custom attribute class. Named properties are listed after all of the parameters required by the constructor. For example

```
[TCustomCodeAttribute(1024, prop1=512, prop2=128)]
TMyClass = class(TObject)
...
end;
```

applies the custom attribute declared above to the class `TMyClass`.

The first parameter, 1024, is the value required by the constructor. The second two parameters are the properties defined in the custom attribute.

When a custom attribute is placed before a list of multiple variable declarations, the attribute applies to all variables declared in that list. For example

```
var
  [TCustomAttribute(1024, prop1=512, prop2=128)]
  x, y, z: Integer;
```

would result in `TCustomAttribute` being applied to all three variables, `x`, `y`, and `z`.

Custom attributes applied to types can be detected at runtime with the `GetCustomAttributes` method of the `Type` class. The following Delphi code demonstrates how to query for custom attributes at runtime.

```
var
  F: TMyClass;           // TMyClass declared above
  T: System.Type;
  A: array of TObject;  // Will hold custom attributes
  I: Integer;

begin
  F := TMyClass.Create;
  T := F.GetType;
  A := T.GetCustomAttributes(True);

  // Write the type name, and then loop over custom
  // attributes returned from the call to
  // System.Type.GetCustomAttributes.
  Writeln(T.FullName);
  for I := Low(A) to High(A) do
    Writeln(A[I].GetType.FullName);
end.
```

Using the DllImport Custom Attribute

You can call unmanaged Win32 APIs (and other unmanaged code) by prefixing the function declaration with the `DllImport` custom attribute. This attribute resides in the `System.Runtime.InteropServices` namespace, as shown below:

```
Program HelloWorld2;

// Don't forget to include the InteropServices unit when using the DllImport attribute.
uses System.Runtime.InteropServices;

[DllImport('user32.dll')]
function MessageBeep(uType : LongWord) : Boolean; external;

begin
    MessageBeep(LongWord(-1));
end.
```

Note the `external` keyword is still required, to replace the block in the function declaration. All other attributes, such as the calling convention, can be passed through the `DllImport` custom attribute.

Custom Attributes and Interfaces

Delphi syntax dictates that the GUID (if present) must immediately follow the declaration of an interface. Since the GUID syntax is similar to that of custom attributes, the compiler must be made to know the difference between a custom attribute - which applies to the next declaration - and a GUID specifier, which applies to the previous declaration. Without this special case, the compiler would try to apply an attribute to the first member of the interface.

When the compiler sees an interface declaration, the next square bracket construct found is assumed to be that of a GUID specifier for the interface. The GUID must be in the traditional Delphi form:

```
[ '{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}' ]
```

Alternatively, you can use the `Guid` custom attribute of the .NET framework (`GuidAttribute`). If you choose this method, then you should introduce the attribute before the interface, as with any other custom attribute.

The effect in either case is the same: the GUID is emitted into the metadata for the interface type. Note that GUIDs are not required for interfaces in the .NET Framework. They are only used for COM interoperability.

Note: When importing COM interfaces with the `ComImport` custom attribute, you must declare the `GuidAttribute` instead of using the Delphi syntax.

C++ Language Guide

This sections contains C++ language topics.

In This Section

[Lexical Elements](#)

[Language Structure](#)

[C++ Specifics](#)

[The Preprocessor](#)

[Keywords, By Category](#)

[Keywords, Alphabetical Listing](#)

Lexical Elements

This section contains Lexical Element topics.

In This Section

[Lexical Elements](#)

[Whitespace Overview](#)

[Tokens Overview](#)

Lexical Elements

These topics provide a formal definition of C++ lexical elements. They describe the different categories of word-like units (tokens) recognized by a language.

The tokens in a C++ source file are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

The preprocessor first scans the program text for special preprocessor directives (see Preprocessor directives for details). For example, the directive `#include <inc_file>` adds (or includes) the contents of the file `inc_file` to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

A C++ program starts as a sequence of ASCII characters representing the source code, created using a suitable text editor (such as the IDE's editor). The basic program unit in C++ is a source file (usually designated by a ".c", or ".cpp" in its name), and all of the header files and other source files included with the `#include` preprocessor directive. Source files are usually designated by a ".c" or ".cpp" in the name, while header files are usually designated with a ".h" or ".hpp".

In the tokenizing phase of compilation, the source code file is parsed (that is, broken down) into tokens and whitespace.

Whitespace Overview

This section contains Whitespace Overview topics.

In This Section

[Whitespace](#)

[Comments](#)

Whitespace

Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i;  
    float f;
```

are lexically equivalent and parse identically to give the six tokens:

- int
- i
- ;
- float
- f
- ;

The ASCII characters representing whitespace can occur within literal strings, in which case they are protected from the normal parsing process (they remain as part of the string). For example,

```
char name[] = "Borland Software Corporation";
```

parses to seven tokens, including the single literal-string token "Borland Software Corporation"

Line splicing with \

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \  
Software Corporation"
```

is parsed as "Borland Software Corporation!" (see String constants for more information).

Comments

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. The compiler supports both methods, and provides an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

You should also follow the guidelines on the use of whitespace and delimiters in comments discussed later in this topic to avoid other portability problems.

C comments

A C comment is any sequence of characters placed after the symbol pair `/*`. The comment terminates at the first occurrence of the pair `*/` following the initial `/*`. The entire sequence, including the four comment-delimiter symbols, is replaced by one space after macro expansion. Note that some C implementations remove comments without space replacements.

The compiler does not support the nonportable token pasting strategy using `/**/`. Token pasting is performed with the ANSI-specified pair `##`, as follows:

```
#define VAR(i,j) (i/**/j)    /* won't work */
#define VAR(i,j) (i##j)     /* OK */
#define VAR(i,j) (i ## j)   /* Also OK */
```

The compiler parses the declaration,

```
int /* declaration */ i /* counter */;
```

as these three tokens:

```
int  i;
```

See Token Pasting with `##` for a description of token pasting.

C++ comments

C++ allows a single-line comment using two adjacent slashes (`//`). The comment can start in any position, and extends until the next new line:

```
class X { // this is a comment
... };
```

You can also use `//` to create comments in C code. This feature is specific to the Borland C++ compiler and is generally not portable.

Nested comments

ANSI C doesn't allow nested comments. The attempt to comment out a line

```
/* int /* declaration */ i /* counter */; */
```

fails, because the scope of the first `/*` ends at the first `*/`. This gives

```
i ; */
```

which would generate a syntax error.

To allow nested comments, check Project|Options|Advanced Compiler|Source|Nested Comments.

Delimiters and whitespace

In rare cases, some whitespace before `/*` and `//`, and after `*/`, although not syntactically mandatory, can avoid portability problems. For example, this C++ code:

```
int i = j/* divide by k*/k;  
+m;
```

parses as `int i = j + m;` not as

```
int i = j/k;  
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;  
+m;
```

avoids this problem.

Tokens Overview

This section contains Token topics.

In This Section

[Tokens](#)

[Keywords Overview](#)

[Identifiers Overview](#)

[Constants Overview](#)

[Punctuators Overview](#)

Tokens

Tokens are word-like units recognized by a language. The compiler recognizes six classes of tokens.

Here is the formal definition of a token:

- keyword
- identifier
- constant
- string-literal
- operator
- punctuator (also known as separators)

As the source code is scanned, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, `external` would be parsed as a single identifier, rather than as the keyword `extern` followed by the identifier `al`.

See Token Pasting with `##` for a description of token pasting.

Keywords Overview

This section contains Keyword topics.

In This Section

[Keywords](#)

[C++-Specific Keywords](#)

[Table Of Borland C++ Register Pseudovariab](#)

[Keyword Extensions](#)

Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names.

If you use non-ANSI keywords in a program and you want the program to be ANSI compliant, always use the non-ANSI keyword versions that are prefixed with double underscores. Some keywords have a version prefixed with only one underscore; these keywords are provided to facilitate porting code developed with other compilers. For ANSI-specified keywords there is only one version.

Note: Note that the keywords `__try` and `try` are an exception to the discussion above. The keyword `try` is required to match the catch keyword in the C++ exception-handling mechanism. `try` cannot be substituted by `__try`. The keyword `__try` can only be used to match the `__except` or `__finally` keywords. See the discussions on C++ exception handling and C-based structured exceptions under Win32 for more information.

Please see the Help table of contents for a complete categorical and alphabetical listing of keywords.

C++-Specific Keywords

A number of keywords are specific to C++ and are not available if you are writing a program in C only. Please see the Help table of contents for a complete categorical and alphabetical listing of these and other keywords.

Table Of Borland C++ Register Pseudovariab

each of these should be jumps

<u>_AH</u>	<u>_CL</u>	<u>_EAX</u>	<u>_ESP</u>
<u>_AL</u>	<u>_CS</u>	<u>_EBP</u>	<u>_FLAGS</u>
<u>_AX</u>	<u>_CX</u>	<u>_EBX</u>	<u>_FS</u>
<u>_BH</u>	<u>_DH</u>	<u>_ECX</u>	<u>_GS</u>
<u>_BL</u>	<u>_DI</u>	<u>_EDI</u>	<u>_SI</u>
<u>_BP</u>	<u>_DL</u>	<u>_EDX</u>	<u>_SP</u>
<u>_BX</u>	<u>_DS</u>	<u>_ES</u>	<u>_SS</u>
<u>_CH</u>	<u>_DX</u>	<u>_ESI</u>	

All but the `_FLAGS` and `_EFLAGS` register pseudovariab

Use register pseudovariab

The flags registers contain information about the state of the 80x86 and the results of recent instructions.

Keyword Extensions

Borland C++ provides a number of keywords that are not part of the ANSI Standard.

Please see the Help table of contents for a complete categorical and alphabetical listing of Library Routines.

Identifiers Overview

This section contains Identifier topics.

In This Section

[Identifiers](#)

Identifiers

Here is the formal definition of an identifier:

identifier:

- nondigit
- identifier nondigit
- identifier digit
- nondigit: one of
- a b c d e f g h i j k l m n o p q r s t u v w x y z _
- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- digit: one of
- 0 1 2 3 4 5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. (Identifiers can contain the letters a to z and A to Z, the underscore character "_", and the digits 0 to 9.) There are only two restrictions:

- The first character must be a letter or an underscore.
- By default, the compiler recognizes only the first 250 characters as significant. The number of significant characters can be reduced by menu and command-line options, but not increased. To change the significant character length, use the spin control in Project|Options|Advanced Compiler|Source|Identifier Length.

Case sensitivity

Identifiers in C and C++ are case sensitive, so that Sum, sum and suM are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, you have the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. With the case-insensitive option, the globals Sum and sum are considered identical, resulting in a possible. "Duplicate symbol" warning during linking.

An exception to these rules is that identifiers of type __pascal are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same scope and sharing the same name space. Duplicate names are legal for different name spaces regardless of scope rules.

Constants Overview

This section contains Constant topics.

In This Section

[Constants](#)

[Integer Constants](#)

[_int8, _int16, _int32, _int64, Unsigned _int64, Extended Integer Types](#)

[Integer Constant Without L Or U](#)

[Floating Point Constants](#)

[Character Constants Overview](#)

[String Constants](#)

[Enumeration Constants](#)

[Constants and Internal Representation](#)

[Internal Representation of Numerical Types](#)

[Constant Expressions](#)

Constants

Constants are tokens representing fixed numeric or character values.

The compiler supports four classes of constants: integer, floating point, character (including strings), and enumeration.

Internal representation of numerical types shows how these types are represented internally.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in the following table.

Constants: Formal Definitions

constant:	nonzero-digit: one of
floating-constant	1 2 3 4 5 6 7 8 9
integer-constant	
numeration-constant	
character-constant	
floating-constant:	octal-digit: one of
fractional-constant <exponent-part> <floating-suffix>	0 1 2 3 4 5 6 7
digit-sequence exponent-part <floating-suffix>	
fractional-constant:	hexadecimal-digit: one of
<digit-sequence> . digit-sequence	0 1 2 3 4 5 6 7 8 9
digit-sequence . a b c d e f	A B C D E F
exponent-part:	integer-suffix:
e <sign> digit-sequence	unsigned-suffix <long-suffix>
E <sign> digit-sequence	long-suffix <unsigned-suffix>
sign: one of	unsigned-suffix: one of
+ -	u U
digit-sequence:	long-suffix: one of

digit	I L
digit-sequence digit	
floating-suffix: one of	enumeration-constant:
f F L	identifier
integer-constant:	character-constant
decimal-constant <integer-suffix>	c-char-sequence
octal-constant <integer-suffix>	
hexadecimal-constant <integer-suffix>	
decimal-constant:	c-char-sequence:
nonzero-digit	c-char
decimal-constant digit	c-char-sequence c-char
octal-constant:	c-char:
0	Any character in the source character set
octal-constant octal-digit	except the single-quote ('), backslash (\), or
	newline character escape-sequence.
hexadecimal-constant:	escape-sequence: one of the following
0 x hexadecimal-digit	\" \' \? \\
0 X hexadecimal-digi	t \a \b \f \n
hexadecimal-constant hexadecimal-digit	\o \oo \ooo \r
	\t \v \Xh... \xh...

Integer Constants

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Integer constants without L or U.. Note that the rules vary between decimal and nondecimal constants.

Decimal

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10; /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0; /*decimal 0 = octal 0 */
```

Octal

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 are truncated.

Hexadecimal

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF are truncated.

long and unsigned suffixes

The suffix L (or l) attached to any constant forces the constant to be represented as a long. Similarly, the suffix U (or u) forces the constant to be unsigned. It is unsigned long if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both L and U suffixes on the same constant in any order or case: ul, lu, UL, and so on.

The data type of a constant in the absence of any suffix (U, u, L, or l) is the first of the following types that can accommodate its value:

Decimal	int, long int, unsigned long int
Octal	int, unsigned int, long int, unsigned long int
Hexadecimal	int, unsigned int, long int, unsigned long int

If the constant has a U or u suffix, its data type will be the first of unsigned int, unsigned long int that can accommodate its value.

If the constant has an L or l suffix, its data type will be the first of long int, unsigned long int that can accommodate its value.

If the constant has both u and l suffixes, (ul, lu, Ul, lU, uL, Lu, LU or UL), its data type will be unsigned long int.

Integer constants without L or U summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding L or U suffix has been used.

__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types

Category

Keyword extensions

Description

You can specify the size for integer types. You must use the appropriate suffix when using extended integers.

Type	Suffix	Example	Storage
__int8	i8	__int8 c = 127i8;	8 bits
__int16	i16	__int16 s = 32767i16;	16 bits
__int32	i32	__int32 i = 123456789i32;	32 bits
__int64	i64	__int64 big = 12345654321i64;	64 bits
unsigned __int64	ui64	unsigned __int64 hugeInt = 1234567887654321ui64;	64 bits

Integer Constant Without L Or U

Decimal constants

0 to 32,767	int
32,768 to 2,147,483,647	long
2,147,483,648 to 4,294,967,295	unsigned long
> 4294967295	truncated

Octal constants

00 to 077777	int
010000 to 0177777	unsigned int

02000000 to 017777777777	long
020000000000 to 037777777777	unsigned long
> 037777777777	truncated

Hexadecimal constants

0x0000 to 0x7FFF	int
0x8000 to 0xFFFF	unsigned int
0x10000 to 0x7FFFFFFF	long
0x80000000 to 0xFFFFFFFF	unsigned long
>0xFFFFFFFF	truncated

Floating Point Constants

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- e or E and a signed integer exponent (optional)
- Type suffix: f or F or l or L (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter e (or E) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Here are some examples:

Constant	Value
23.45e6	23.45 (10^6)
.0	0
0.	0.0
1.	1.0
-1.23	-1.23
2e-5	2.0 (10^{-5})
3E+10	3.0 (10^{10})
.09E34	0.09 (10^{34})

In the absence of any suffixes, floating-point constants are of type double. However, you can coerce a floating constant to be of type float by adding an f or F suffix to the constant. Similarly, the suffix l or L forces the constant to be data type long double. The table below shows the ranges available for float, double, and long double.

Floating-point constant sizes and ranges

Type	Size (bits)	Range
float	32	3.4×10^{-38} to 3.4×10^{38}
double	64	1.7×10^{-308} to 1.7×10^{308}
long double	80	3.4×10^{-4932} to 1.1×10^{4932}

Character Constants Overview

This section contains Character Constant topics.

In This Section

[Character Constants](#)

[The Three Char Types](#)

[Escape Sequences](#)

[Wide-character And Multi-character Constants](#)

Character Constants

A character constant is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-character constants have data type int. In C++, a character constant has type char. Multicharacter constants in both C and C++ have data type int.

To learn more about character constants, see the following topics.

- Three char types
- Escape sequences
- Wide-character and multi-character constants

Note: To compare sizes of character types, compile this as a C program and then as a C++ program.

```
#include <stdio.h>
#define CH 'x'      /* A CHARACTER CONSTANT */
void main(void) {
    char ch = 'x';  /* A char VARIABLE */
    printf("\nSizeof int      = %d", sizeof(int) );
    printf("\nSizeof char    = %d", sizeof(char) );
    printf("\nSizeof ch      = %d", sizeof(ch) );
    printf("\nSizeof CH      = %d", sizeof(CH) );
    printf("\nSizeof wchar_t = %d", sizeof(wchar_t) );
}
```

Note: Sizes are in bytes.

Sizes of character types

Output when compiled as C program	Output when compiled as C++ program
Sizeof int = 4	Sizeof int = 4
Sizeof char = 1	Sizeof char = 1
Sizeof ch = 1	Sizeof ch = 1
Sizeof CH = 4	Sizeof CH = 1
Sizeof wchar_t = 2	Sizeof wchar_t = 2

The Three Char Types

One-character constants, such as 'A', '\t' and '007', are represented as int values. In this case, the low-order byte is sign extended into the high bit; that is, if the value is greater than 127 (base 10), the upper bit is set to -1 (=0xFF). This can be disabled by declaring that the default char type is unsigned.

The three character types, char, signed char, and unsigned char, require an 8-bit (one byte) storage. By default, the compiler treats character declarations as signed. Use the -K compiler option to treat character declarations as unsigned. The behavior of C programs is unaffected by the distinction between the three character types.

In a C++ program, a function can be overloaded with arguments of type char, signed char, or unsigned char. For example, the following function prototypes are valid and distinct:

```
void func(char ch);
void func(signed char ch);
void func(unsigned char ch);
```

If only one of the above prototypes exists, it will accept any of the three character types. For example, the following is acceptable:

```
void func(unsigned char ch);
void main(void)
{
    signed char ch = 'x';
    func(ch);
}
```

Escape Sequences

The backslash character (\) is used to introduce an escape sequence, which allows the visual representation of certain nongraphic characters. For example, the constant \n is used to the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for Ctrl-C or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type char (0 to 0xff). Larger numbers generate the compiler error Numeric constant too large. For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Take this example.

```
printf("\x0072.1A Simple Operating System");
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, the compiler treats it as the hexadecimal number \x0072 and the literal string "2.1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

The following table shows the available escape sequences.

Escape sequences

Note: You must use \\ to represent an ASCII backslash, as used in operating system paths.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O=a string of up to three octal digits
\xH		any	H=a string of hex digits
\XH		any	H=a string of hex digits

Wide-character And Multi-character Constants

Wide-character types can be used to represent a character that does not fit into the storage space allocated for a char type. A wide character is stored in a two-byte space. A character constant preceded immediately by an L is a wide-character constant of data type wchar_t (defined in stddef.h). For example:

```
wchar_t ch = L'A';
```

When wchar_t is used in a C program it is a type defined in stddef.h header file. In a C++ program, wchar_t is a keyword that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In Borland C++, wchar_t is the same size, signedness, and alignment requirement as an unsigned short type.

A string preceded immediately by an L is a wide-character string. The memory allocation for a string is two bytes per character. For example:

```
wchar_t *str = L"ABCD";
```

Multi-character constants

The compiler also supports multi-character constants. Multi-character constants can consist of as many as four characters. For example, the constant, '\006\007\008\009' is valid only in a Borland C++ program. Multi-character constants are always 32-bit int values. The constants are not portable to other C++ compilers.

String Constants

This section contains String Constant topics.

In This Section

[String Constants](#)

String Constants

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of- `constchar` and storage class `static`, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written `""`.

The characters inside the double quotes can include escape sequences. This code, for example:

```
"\t\t\"Name\"\\\"Address\n\n"
```

prints like this:

```
"Name" \      Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The `\` provides interior double quotes.

If you compile with the `-A` option for ANSI compatibility, the escape character sequence `"\"`, is translated to `"\"` by the compiler.

A literal string is stored internally as the given sequence of characters plus a final null character (`'\0'`). A null string is stored as a single `'\0'` character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>
int main() {
    char    *p;
    p = "This is an example of how the compiler " " will\nconcatenate very long strings for
you" " automatically, \nresulting in nicer" " looking programs.";
    printf(p);
    return(0);
}
```

The output of the program is

```
This is an example of how the compiler will
concatenate very long strings for you automatically,
resulting in nicer looking programs.
```

You can also use the backslash (`\`) as a continuation character to extend a string constant across line boundaries:

```
puts("This is really \  
a one-line string");
```

Enumeration Constants

This section contains Enumeration Constant topics.

In This Section

[Enumeration Constants](#)

Enumeration Constants

Enumeration constants are identifiers defined in enum type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the enum declaration. Negative initializers are allowed. See Enumerations and enum (keyword) for a detailed look at enum declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional initializers. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, cubs, and dodgers are enumeration constants of type team that can be assigned to any variables of type team or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```


Constants and Internal Representation

This section contains Constants and Internal Representation topics.

In This Section

[Constants And Internal Representation](#)

Constants And Internal Representation

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For Borland C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 80x86 and the Pentium family of microprocessors governs the choices of internal representations for the various data types.

The following tables list the sizes and resulting ranges of the data types. Internal representation of numerical types shows how these types are represented internally.

32-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	$0 \leq X \leq 255$	Small numbers and full PC character set
char	8	$-128 \leq X \leq 127$	Very small numbers and ASCII characters
short int	16	$-32,768 \leq X \leq 32,767$	Counting, small numbers, loop control
unsigned int	32	$0 \leq X \leq 4,294,967,295$	Large numbers and loops
int	32	$-2,147,483,648 \leq X \leq 2,147,483,647$	Counting, small numbers, loop control
unsigned long	32	$0 \leq X \leq 4,294,967,295$	Astronomical distances
enum	32	$-2,147,483,648 \leq X \leq 2,147,483,647$	Ordered sets of values
long	32	$-2,147,483,648 \leq X \leq 2,147,483,647$	Large numbers, populations
float	32	$1.18 (10^{-38} < X < 3.40 (10^{38}$	Scientific (7-digit) precision)
double	64	$2.23 (10^{-308} < X < 1.79 (10^{308}$	Scientific (15-digit precision)
long double	80	$3.37 (10^{-4932} < X < 1.18 (10^{4932}$	Financial (18-digit precision)

Internal Representation of Numerical Types

This section contains Internal Representation of Numerical Type topics.

In This Section

[Internal Representation Of Numerical Types](#)

Internal Representation Of Numerical Types

32-bit integers

Floating-point types, always

s	= Sign bit (0 = positive, 1 = negative)	Exponent bias (normalized values):
i	= Position of implicit binary point	float: 127 (7FH)
1	= Integer bit of significance:	double: 1,023 (3FFH)
	Stored in long doubleImplicit in float, double	long double: 16,383 (3FFFH)

Constant Expressions

This section contains Constant Expression topics.

In This Section

[Constant Expressions](#)

Constant Expressions

A constant expression is an expression that always evaluates to a constant (it is evaluated at compile-time and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is:

```
constant-expression:  
Conditional-expression
```

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a sizeof operator:

- Assignment
- Comma
- Decrement
- Function call
- Increment

Punctuators Overview

This section contains Punctuator topics.

In This Section

[Punctuators](#)

Punctuators

The C++ punctuators (also known as separators) are:

- []
- ()
- {}
- ,
- ;
- :
- ...
- *
- =
- #

Most of these punctuators also function as operators.

Brackets

Open and close brackets indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];          /* 3 x 4 matrix */
ch = str[3];            /* 4th element */
.
.
.
```

Parentheses

Open and close parentheses () are used to group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b); /* override normal precedence */
if (d == z) ++x; /* essential with conditional statement */
func(); /* function call, no args */
int (*fptr)(); /* function pointer declaration */
fptr = func; /* no () means func pointer */
void func2(int n); /* function declaration with parameters */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered in Expressions.

Braces

Open and close braces { } indicate the start and end of a compound statement:

```
if (d == z)
{
    ++x;
    func();
}
```

The closing brace serves as a terminator for the compound statement, so a ; (semicolon) is not required after the }, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
    {};          /* illegal semicolon */
else
```

Comma

The comma (,) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them. Note that (exp1, exp2) evaluates both but is equal to the second:

```
func(i, j);          /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5)); /* also calls func with two args! */
```

Semicolon

The semicolon (;) is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an expression statement. The expression is evaluated and its value is discarded. If the expression statement has no side effects, the compiler might ignore it.

```
a + b;    /* maybe evaluate a + b, but discard value */
++a;      /* side effect on a, but discard value of ++a */
;         /* empty expression = null statement */
```

Semicolons are often used to create an empty statement:

```
for (i = 0; i < n; i++)
{
    ;
}
```

Colon

Use the colon (:) to indicate a labeled statement:

```
start:    x=0;
...
goto start;
```

Labels are discussed in Labeled statements.

Ellipsis

The ellipsis (...) is three successive periods with no intervening whitespace. Ellipses are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that func will be defined in such a way that calls must have at least two arguments, an int and a char, but can also have any number of additional arguments.

In C++, you can omit the comma before the ellipsis.

Asterisk (pointer declaration)

The asterisk (*) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr; /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;      /* a pointer to an integer array */  
double ***double_ptr; /* a pointer to a matrix of doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;  
a = b * 3.14;
```

Equal sign (initializer)

The equal sign (=) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };  
int x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... } /* Parameter i has default value of zero */
```

The equal sign is also used as the assignment operator in expressions:

```
int a, b, c;  
a = b + c;  
float *ptr = (float *) malloc(sizeof(float) * 100);
```

Pound sign (preprocessor directive)

The pound sign (#) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See Preprocessor directives for more on the preprocessor directives.

and ## (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase. See Token pasting.

Language Structure

This section contains Language Structure topics.

In This Section

[Language Structure](#)

[Declarations](#)

[Declaration Syntax](#)

[Pointers](#)

[Arrays](#)

[Functions](#)

[Structures](#)

[Unions](#)

[Enumerations](#)

[Expressions](#)

[Operators Summary](#)

[Primary Expression Operators](#)

[Postfix Expression Operators](#)

[Unary Operators](#)

[Binary Operators](#)

[Statements](#)

Language Structure

These topics provide a formal definition of C++ language and its implementation in the Borland C++ compiler. They describe the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

Declarations

This section contains Declaration topics.

In This Section

[Declarations](#)

[Objects](#)

[Storage Classes And Types](#)

[Scope](#)

[Visibility](#)

[Duration](#)

[Translation Units](#)

[Linkage](#)

Declarations

This section briefly reviews concepts related to declarations: objects, storage classes, types, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Objects

This section contains Object topics.

In This Section

[Objects](#)

Objects

An object is a specific region of memory that can hold a fixed or variable value (or set of values). (This use of the word object is different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a data type). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely references the object. The type is used

- to determine the correct memory allocation requirements.
- to interpret the bit patterns found in the object during subsequent accesses.
- in many type-checking situations, to ensure that illegal assignments are trapped.

Borland's C++ compiler supports all standard data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in memory.

Objects and declarations

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as defining declarations, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Other declarations, known as referencing declarations, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its declaration point in the source code. Legal exceptions to this rule (known as forward references) are labels, calls to undeclared functions, and class, struct, or union tags.

lvalues

An lvalue is an object locator: an expression that designates an object. An example of an lvalue expression is *P, where P is any expression evaluating to a non-null pointer. A modifiable lvalue is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A const pointer to a constant, for example, is not a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the l stood for "left," meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if a and b are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as a = 1; and b = a + b are legal.

rvalues

The expression a + b is not an lvalue: a + b = a is illegal because the expression on the left is not related to an object. Such expressions are often called rvalues (short for right values).

Storage Classes And Types

This section contains Storage Classes and Type topics.

In This Section

[Storage Classes And Types](#)

Storage Classes And Types

Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The compile-time operator, `sizeof`, lets you determine the size in bytes of any standard or user-defined type. See `sizeof` for more on this operator.

Scope

This section contains Scope topics.

In This Section

[Scope](#)

Scope

The scope of an identifier is that part of the program in which the identifier can be used to access its object. There are six categories of scope: block (or local), function, function prototype, file, class (C++ only), condition (C++ only), and namespace (C++ only). These depend on how and where identifiers are declared.

- **Block.** The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the enclosing block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.
- **Function.** The only identifiers having function scope are statement labels. Label names can be used with goto statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing label_name: followed by a statement. Label names must be unique within a function.
- **Function prototype.** Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.
- **File.** File scope identifiers, also known as globals, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.
- **Class (C++).** A class is a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see [Classes](#).
- **Condition (C++).** Declarations in conditions are supported. Variables can be declared within the expression of if, while, and switch statements. The scope of the variable is that of the statement. In the case of an if statement, the variable is also in scope for the else block.
- **namespace (C++).** A namespace is a logical grouping of program entities (e.g. identifiers, classes, and functions). Namespaces are open, that is, they can span multiple compilation units. You can think of a namespace as introducing a named scope, similar in many ways to a class in C++. See the help for the namespace keyword for more information on how to declare and use namespaces.

Name spaces

Name space is the scope within which an identifier must be unique. Note that a C++ namespace extends this concept by allowing you to give the scope a name. In addition to the named-scoping capability of C++, the C programming language uses four distinct classes of identifiers:

- **goto label names.** These must be unique within the function in which they are declared.
- **Structure, union, and enumeration tags.** These must be unique within the block in which they are defined. Tags declared outside of any function must be unique.
- **Structure and union member names.** These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
- **Variables, typedefs, functions, and enumeration members.** These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Visibility

This section contains Visibility topics.

In This Section

[Visibility](#)

Visibility

The visibility of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Note: Visibility cannot exceed scope, but scope can exceed visibility.

Again, special rules apply to hidden class names and class member names: C++ operators allow hidden identifiers to be accessed under certain conditions

Duration

This section contains Duration topics.

In This Section

[Duration](#)

[Static](#)

Duration

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike typedefs and types, have real memory allocated during run time. There are three kinds of duration: static, local, and dynamic.

Static

Memory is allocated to objects with static duration as soon as execution is underway; this storage allocation lasts until the program terminates. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit static or extern storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, a class constructor.

Don't confuse static duration with file or global scope. An object can have static duration and local scope.

Local

Local duration objects, also known as automatic objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects must always have local or function scope. The storage class specifier auto can be used when declaring local duration variables, but is usually redundant, because auto is the default for variables declared within a block. An object with local duration also has local scope, because it does not exist outside of its enclosing block. The converse is not true: a local scope object can have static duration.

When declaring variables (for example, int, char, float), the storage class specifier register also implies auto; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. The compiler can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an auto, local object with no warning or error.

Note: The compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

Dynamic

Dynamic duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the heap, using either standard library functions such as malloc, or by using the C++ operator new. The corresponding deallocations are made using free or delete.

Static

Category

Storage class specifiers

Syntax

```
static <data definition> ;static  
static <function name> <function definition> ;
```

Description

Use the static storage class specifier with a local variable to preserve the last value between successive calls to that function. A static variable acts like a local variable but has the lifetime of an external variable.

In a class, data and member functions can be declared static. Only one copy of the static data exists for all objects of the class.

A static member function of a global class has external linkage. A member of a local class has no linkage. A static member function is associated only with the class in which it is declared. Therefore, such member functions cannot be virtual.

Static member functions can only call other static member functions and only have access to static data. Such member functions do not have a this pointer.

Translation Units

This section contains Translation Unit topics.

In This Section

[Translation Units](#)

Translation Units

The term translation unit refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

```
translation-unit:  
external-declaration  
translation-unit external-declaration  
external-declaration  
function-definition  
declaration
```

word external has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the section Linkage..) Any declaration that also reserves storage for an object or function is called a definition (or defining declaration). For more details, see External declarations and definitions.

Linkage

This section contains Linkage topics.

In This Section

[Linkage](#)

Linkage

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier `static` or `extern`.

Each instance of a particular identifier with external linkage represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with internal linkage represents the same object or function within one file only. Identifiers with no linkage represent unique entities.

External and internal linkage rules

Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier `static`.

For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.

If the declaration of an object or function identifier contains the storage class specifier `extern`, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.

If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier `extern` had been used.

If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

Identifiers with no linkage attribute:

- Any identifier declared to be other than an object or a function (for example, a typedef identifier)
- Function parameters
- Block scope identifiers for objects declared without the storage class specifier `extern`

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or `.obj` files compiled with a C compiler

To tell the C++ compiler not to mangle the name of a function, declare the function as `extern "C"`, like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler not to mangle references to the function `Cfunc`.

You can also apply the extern "C" declaration to a block of names:

```
extern "C" {  
    void Cfunc1( int );  
    void Cfunc2( int );  
    void Cfunc3( int );  
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions Cfunc1, Cfunc2, and Cfunc3 should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {  
    #include "locallib.h"  
};
```

Note: extern "C" cannot be used with class identifiers.

Declaration Syntax

This section contains Declaration Syntax topics.

In This Section

- [Declaration Syntax](#)
- [Tentative Definitions](#)
- [Possible Declarations](#)
- [External Declarations And Definitions](#)
- [Type Specifiers](#)
- [Type Categories](#)
- [The Fundamental Types](#)
- [Initialization](#)
- [Declarations And Declarators](#)
- [Use Of Storage Class Specifiers](#)
- [Variable Modifiers](#)
- [Mixed-Language Calling Conventions](#)
- [Multithread Variables](#)
- [Function Modifiers](#)

Declaration Syntax

All six interrelated attributes (storage classes, types, scope, visibility, duration, and linkage) are determined in diverse ways by declarations.

Declarations can be defining declarations (also known as definitions) or referencing declarations (sometimes known as nondefining declarations). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Tentative Definitions

The ANSI C standard supports the concept of the tentative definition. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the extern storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;
int x;          /*legal, one copy of x is reserved */
int y;
int y = 4;      /* legal, y is initialized to 4 */
int z = 5;
int z = 6;      /* not legal, both are initialized definitions */
```

Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

Possible Declarations

The range of objects that can be declared includes

- Variables
- Functions
- Classes and class members (C++)
- Types
- Structure, union, and enumeration tags
- Structure members
- Union members
- Arrays of other types
- Enumeration constants
- Statement labels
- Preprocessor macros

The full syntax for declarations is shown in Tables 2.1 through 2.3. The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use typedefs to improve legibility.

In Borland C++ declaration syntax., note the restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail in Variable Modifiers and Function Modifiers.

Borland C++ declaration syntax

declaration:	elaborated-type-specifier:
<decl-specifiers> <declarator-list>;	class-key identifier
asm-declaration	class-key class-name
function-declaration	enum enum-name
linkage-specification	class-key: (C++ specific)
decl-specifier:	class
storage-class-specifier	struct
type-specifier	union
function-specifier	enum-specifier:
friend (C++ specific)	enum <identifier> { <enum-list> }
typedef	enum-list:
decl-specifiers:	enumerator
<decl-specifiers> decl-specifier	enumerator-list , enumerator
storage-class-specifier:	enumerator:
auto	identifier
register	identifier = constant-expression
static	constant-expression:
extern	conditional-expression
function-specifier: (C++ specific)	linkage-specification: (C++ specific)
inline	extern string { <declaration-list> }
virtual	extern string declaration
simple-type-name:	type-specifier:
class-name	simple-type-name
typedef-name	class-specifier
boolean	
char	enum-specifier
short	elaborated-type-specifier

int	const
__int8	
__int16	
__int32	
__int64	
long	volatile
signed	declaration-list:
unsigned	declaration
float	declaration-list ; declaration
double	
void	
declarator-list:	type-name:
init-declarator	type-specifier <abstract-declarator>
declarator-list , init-declarator	abstract-declarator:
init-declarator:	pointer-operator <abstract-declarator>
declarator <initializer>	<abstract-declarator> (argument-declaration-list)
declarator:	<cv-qualifier-list>
dname	<abstract-declarator> [<constant-expression>]
modifier-list	(abstract-declarator)
pointer-operator declarator	argument-declaration-list:
declarator (parameter-declaration-list)	<arg-declaration-list>
<cv-qualifier-list >	arg-declaration-list , ...
(The <cv-qualifier-list > is for C++ only.)	<arg-declaration-list> ... (C++ specific)
declarator [<constant-expression>]	arg-declaration-list:
(declarator)	argument-declaration
modifier-list:	arg-declaration-list , argument-declaration
modifier	argument-declaration:
modifier-list modifier	decl-specifiers declarator
modifier:	decl-specifiers declarator = expression
__cdecl	(C++ specific)
__pascal	decl-specifiers <abstract-declarator>
__stdcall	decl-specifiers <abstract-declarator> = expression
__fastcall	(C++ specific)
function-definition:	
function-body:	
pointer-operator:	compound-statement
* <cv-qualifier-list>	initializer:
& <cv-qualifier-list> (C++ specific)	= expression
class-name :: * <cv-qualifier-list>	= { initializer-list }
(C++ specific)	(expression-list) (C++ specific)
cv-qualifier-list:	initializer-list:
cv-qualifier <cv-qualifier-list>	expression
cv-qualifier	initializer-list , expression
const	{ initializer-list <, > }
volatile	
dname:	
name	
class-name (C++ specific)	
~ class-name (C++ specific)	
type-defined-name	

External Declarations And Definitions

The storage class specifiers `auto` and `register` cannot appear in an external declaration. For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of `sizeof`), then exactly one external definition of that identifier must exist in the entire program.

The C++ compiler allows later declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. Here's an example:

```
int a[];           // no size
struct mystruct;   // tag only, no member declarators
.
.
.
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
    int i, j;
};                // add member declarators
```

Borland C++ class declaration syntax (C++ only) covers class declaration syntax. In the section on classes (beginning with [Classes](#)), you can find examples of how to declare a class. Referencing covers C++ reference types (closely related to pointer types) in detail. Finally, see [Using Templates](#) for a discussion of template-type classes.

Borland C++ class declaration syntax (C++ only)

class-specifier: base-specifier:

class-head { <member-list> } : base-list

class-head: base-list:

class-key <identifier> <base-specifier> base-specifier

class-key class-name <base-specifier> base-list , base-specifier

member-list: base-specifier:

member-declaration <member-list> class-name

access-specifier : <member-list> virtual <access-specifier> class-name

member-declaration: access-specifier <virtual> class-name

<decl-specifiers> <member-declarator-list> ; access-specifier:

function-definition <;> private

qualified-name ; protected

member-declarator-list: public

member-declarator conversion-function-name:

member-declarator-list, member-declarator operator conversion-type-name

member-declarator: conversion-type-name:

declarator <pure-specifier> type-specifiers <pointer-operator>

<identifier> : constant-expression constructor-initializer:

pure-specifier: : member-initializer-list

= 0

member-initializer-list: operator-name: one of
 member-initializer new delete sizeof typeid
 member-initializer , member-initializer-list +-* /%^
 member-initializer: &|~!=<>
 class name (<argument-list>) +-=**/= %= ^=
 identifier (<argument-list>) &|= <<>>>=<<=
 operator-function-name: ==!=<=>=&&||
 operator operator-name ++ __, ->*->()
 [].*

Type Specifiers

The type determines how much memory is allocated to an object and how the program interprets the bit patterns found in the object's storage allocation. A data type is the set of values (often implementation-dependent) identifiers can assume, together with the set of operations allowed on those values.

The type specifier with one or more optional modifiers is used to specify the type of the declared identifier:

```
int i;           // declare i as an integer
unsigned char ch1, ch2; // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type signed int (or equivalently, int) is the assumed default. However, in C++, a missing type specifier can lead to syntactic ambiguity, so C++ practice requires you to explicitly declare all int type specifiers.

The type specifier keywords by the Borland C++ compiler are:

char	float	signed
wchar_t		
class	int	struct
double	long	union
enum	short	unsigned

Use the sizeof operators to find the size in bytes of any predefined or user-defined type.

Type Categories

This section contains Type Category topics.

In This Section

[Type Categories](#)

[Void](#)

Type Categories

The four basic type categories (and their subcategories) are as follows:

Aggregate

Array

struct

union

class (C++ only)

Function

Scalar

Arithmetic

Enumeration

Pointer

Reference (C++ only)

void

Types can also be viewed in another way: they can be fundamental or derived types. The fundamental types are void, char, int, float, and double, together with short, long, signed, and unsigned variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.

A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes

Given any nonvoid type type (with some provisos), you can declare derived types as follows:

Declaring types

Declaration	Description
type t;	An object of type type
type array[10];	Ten types: array[0] - array[9]
type *ptr;	ptr is a pointer to type
type &ref = t;	ref is a reference to type (C++)
type func(void);	func returns value of type type
void func1(type t);	func1 takes a type type parameter
struct st {type t1;type t2};	structure st holds two types

Note: type& var, type &var, and type & var are all equivalent.

Void

Category

Special types

Syntax

```
void identifier
```

Description

void is a special type indicating the absence of any value. Use the void keyword as a function return type if the function does not return a value.

```
void hello(char *name)
{
    printf("Hello, %s.",name);
}
```

Use void as a function heading if the function does not take any parameters.

```
int init(void)
{
    return 1;
}
```

Void Pointers

Generic pointers can also be declared as void, meaning that they can point to any type.

void pointers cannot be dereferenced without explicit casting because the compiler cannot determine the size of the pointer object.

The Fundamental Types

This section contains Fundamental Type topics.

In This Section

[The Fundamental Types](#)

The Fundamental Types

The fundamental type specifiers are built from the following keywords:

char	__int8	long
double	__int16	signed
float	__int32	short
int	__int64	unsigned

From these keywords you can build the integral and floating-point types, which are together known as the arithmetic types. The modifiers long, short, signed, and unsigned can be applied to the integral types. The include file limits.h contains definitions of the value ranges for all the fundamental types.

Integral types

char, short, int, and long, together with their unsigned variants, are all considered integral data types. Integral types shows the integral type specifiers, with synonyms listed on the same line.

Integral types

char, signed char	Synonyms if default char set to signed.
unsigned char	
char, unsigned char	Synonyms if default char set to unsigned.
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Note: These synonyms are not valid in C++. See The three char types.

signed or unsigned can only be used with char, short, int, or long. The keywords signed and unsigned, when used on their own, mean signed int and unsigned int, respectively.

In the absence of unsigned, signed is assumed for integral types. An exception arises with char. You can set the default for char to be signed or unsigned. (The default, if you don't set it yourself, is signed.) If the default is set to unsigned, then the declaration char ch declares ch as unsigned. You would need to use signed char ch to override the default. Similarly, with a signed default for char, you would need an explicit unsigned char ch to declare an unsigned char.

Only long or short can be used with int. The keywords long and short used on their own mean long int and short int.

ANSI C does not dictate the sizes or internal representations of these types, except to indicate that short, int, and long form a nondecreasing sequence with "short <= int <= long." All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

The compiler regards the types int and long as equivalent, both being 32 bits. The signed varieties are all stored in two's complement format using the most significant bit (MSB) as a sign bit: 0 for positive, 1 for negative (which explains the ranges shown in 32-bit data types, sizes, and ranges). In the unsigned versions, all bits are used to give a range of 0 - (2ⁿ - 1), where n is 8, 16, or 32.

Floating-point types

The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. The compiler uses the IEEE floating-point formats. See the topic on ANSI implementation-specific.

float and double are 32- and 64-bit floating-point data types, respectively. long can be used with double to declare an 80-bit precision floating-point identifier: long double test_case, for example.

The table of 32-bit data types, sizes, and ranges indicates the storage allocations for the floating-point types

Standard arithmetic conversions

When you use an arithmetic expression, such as a + b, where a and b are different arithmetic types, The compiler performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps the compiler uses to convert the operands in an arithmetic expression:

- 1. Any small integral types are converted as shown in Methods used in standard arithmetic conversions. After this, any two values associated with an operator are either int (including the long and unsigned modifiers), or they are of type double, float, or long double.
- 2. If either operand is of type long double, the other operand is converted to long double.
- 3. Otherwise, if either operand is of type double, the other operand is converted to double.
- 4. Otherwise, if either operand is of type float, the other operand is converted to float.
- 5. Otherwise, if either operand is of type unsigned long, the other operand is converted to unsigned long.
- 6. Otherwise, if either operand is of type long, then the other operand is converted to long.
- 7. Otherwise, if either operand is of type unsigned, then the other operand is converted to unsigned.
- 8. Otherwise, both operands are of type int.

The result of the expression is the same type as that of the two operands.

Methods used in standard arithmetic conversions

Type	Converts to	Method
char	int	Zero or sign-extended (depends on default char type)
unsigned char	int	Zero-filled high byte (always)
signed char	int	Sign-extended (always)
short	int	Same value; sign extended
unsigned short	unsigned int	Same value; zero filled
enum	int	Same value

Special char, int, and enum conversions

Note: The conversions discussed in this section are specific to the Borland C++ compiler.

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type signed char always use sign extension; objects of type unsigned char always set the high byte to zero when converted to int.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign-extends or zero-fills the extra bits of the new value, depending on whether the shorter type is signed or unsigned, respectively.

Initialization

This section contains Initialization topics.

In This Section

[Initialization](#)

Initialization

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

- To zero if it is an arithmetic type
- To null if it is a pointer type

Note: If the object has automatic storage duration, its value is indeterminate.

Syntax for initializers

```
initializer
    = expression
    = {initializer-list} <,>
    (expression list)
initializer-list
    expression
    initializer-list, expression
    {initializer-list} <,>
```

Rules governing initializers

The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.

The item to be initialized must be an object (for example, an array).

For C (not required for C++), all expressions must be constants if they appear in one of these places:

In an initializer for an object that has static duration.

In an initializer list for an array, structure, or union (expressions using `sizeof` are also allowed).

- If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
- If a brace-enclosed list has fewer initializers than members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- An initializer list (as described in Arrays, structures, and unions).
- A single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array `days`, which counts how many times each day of the week appears in a month (assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

The following rules initialize character arrays and wide character arrays:

- You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for `name[0]`), 'n' (for `name[1]`), and so on (and including a null terminator).

- You can initialize a wide character array (one that is compatible with `wchar_t`) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {  
    int i;  
    char str[21];  
    double d;  
} s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces.

Declarations And Declarators

This section contains Declarations And Declarator topics.

In This Section

[Declarations And Declarators](#)

Declarations And Declarators

A declaration is a list of names. The names are sometimes referred to as declarators or identifiers. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon.

Simple declarations of variable identifiers have the following pattern:

```
data-type var1 <=init1>, var2 <=init2>, ...;
```

where var1, var2,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type data-type. For example,

```
int x = 1, y = 2;
```

creates two integer variables called x and y (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.

In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions

The format of the declarator indicates how the declared name is to be interpreted when used in an expression. If type is any type, and storage class specifier is any storage class specifier, and if D1 and D2 are any two declarators, then the declaration

```
storage-class-specifier type D1, D2;
```

indicates that each occurrence of D1 or D2 in an expression will be treated as an object of type type and storage class storage class specifier. The type of the name embedded in the declarator will be some phrase containing type, such as "type," "pointer to type," "array of type," "function returning type," or "pointer to function returning type," and so on.

For example, in Declaration syntax examples each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single int object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Declaration syntax examples

Declarator syntax	Implied type of name	Example
type	name; type	int count;
type name[];	(open) array of type	int count[];
type name[3];	Fixed array of three elements, int count[3] all of type;	(name[0], name[1], and name[2])
type *name;	Pointer to type	int *count;
type *name[];	(open) array of pointers to type	int *count[];

type *(name[]);	Same as above	int *(count[]);
type (*name)[];	Pointer to an (open) array of type	int (*count) [];
type &name;	Reference to type (C++ only)	int &count;
type name();	Function returning type	int count();
type *name();	Function returning pointer to type	int *count();
type *(name());	Same as above	int *(count());
type (*name)();	Pointer to function returning type	int (*count) ();

Note the need for parentheses in (*name)[] and (*name)(); this is because the precedence of both the array declarator [] and the function declarator () is higher than the pointer declarator *. The parentheses in *(name[]) are optional.

Note: See Borland C++ declaration syntax for the declarator syntax. The definition covers both identifier and function declarators.

Use Of Storage Class Specifiers

This section contains Use Of Storage Class Specifier topics.

In This Section

[Storage Class Specifiers](#)

Storage Class Specifiers

Storage classes specifiers are also called type specifiers. They dictate the location (data segment, register, heap, or stack) of an object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the declaration syntax, by its placement in the source code, or by both of these factors.

The keyword mutable does not affect the lifetime of the class member to which it is applied.

The storage class specifiers in C++ are:

- auto register
- __declspec static
- extern typedef
- mutable

Variable Modifiers

This section contains Variable Modifier topics.

In This Section

[Variable Modifiers](#)

[Const](#)

[Volatile](#)

Variable Modifiers

In addition to the storage class specifier keywords, a declaration can use certain modifiers to alter some aspect of the identifier. The modifiers available are summarized in Borland C++ modifiers.

Mixed-language calling conventions

You can call routines written in other languages, and vice versa. When you mix languages, you have to deal with two important issues: identifiers and parameter passing.

By default, the compiler saves all global identifiers in their original case (lower, upper, or mixed) with an underscore "_" prepended to the front of the identifier. To remove the default, you can use the -u command-line option.

Note: The section Linkage tells how to use extern, which allows C names to be referenced from a C++ program.

The following table summarizes the effects of a modifier applied to a called function. For every modifier, the table shows the order in which the function parameters are pushed on the stack. Next, the table shows whether the calling program (the caller) or the called function (the callee) is responsible for popping the parameters off the stack. Finally, the table shows the effect on the name of a global function.

Calling conventions

Modifier	Push parameters	Pop parameters	Name change (only in C)
__cdecl	Right to left	Caller	'_' prepended
__fastcall	Left to right	Callee	'@' prepended
__pascal	Left to right	Callee	Uppercase
__stdcall	Right to left	Callee	No change

1. This is the default.

Note: Note: __fastcall and __stdcall are always name mangled in C++. See the description of the -VC option.

Const

Category

Modifiers

Syntax

```
const <variable name> [ = <value> ] ;constconst
<function name> ( const <type>*<variable name> ;)
<function name> const;
```

Description

Use the const modifier to make a variable value unmodifiable.

Use the const modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a const result in a compiler error.

A const pointer cannot be modified, though the object to which it points can be changed. Consider the following examples.

```
const float pi    = 3.14;
const maxint  = 12345;    // When used by itself, const is equivalent to int.
char *const str1 = "Hello, world";    // A constant pointer
char const *str2 = "Borland Software Corporation"; // A pointer to a constant character
string.
```

Given these declarations, the following statements are illegal.

```
pi    = 3.0;    // Assigns a value to a const.
i      = maxint++; // Increments a const.
str1 = "Hi, there!" // Points str1 to something else.
```

Using the const Keyword in C++ Programs

C++ extends const to include classes and member functions. In a C++ class definition, use the const modifier following a member function declaration. The member function is prevented from modifying any data in the class.

A class object defined with the const keyword attempts to use only member functions that are also defined with const. If you call a member function that is not defined as const, the compiler issues a warning that a non-const function is being called for a const object. Using the const keyword in this manner is a safety feature of C++.

Warning: A pointer can indirectly modify a const variable, as in the following:

```
*(int *)&my_age = 35;
```

If you use the const modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int printf (const char *format, ...);
```

printf is prevented from modifying the format string.

Volatile

Category

Modifiers

Syntax

```
volatile <data definition> ;
```

Description

Use the volatile modifier to indicate that a background routine, an interrupt routine, or an I/O port can change a variable. Declaring an object to be volatile warns the compiler not to make assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile int ticks;
void timer( ) {
```



```
ticks++;  
}  
void wait (int interval) {  
    ticks = 0;  
    while (ticks < interval); // Do nothing  
}
```

The routines in this example (assuming timer has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument interval. A highly optimizing compiler might not load the value of ticks inside the test of the while loop since the loop doesn't change the value of ticks.

Note: C++ extends volatile to include classes and member functions. If you've declared a volatile object, you can use only its volatile member functions.

Mixed-Language Calling Conventions

This section contains Mixed-Language Calling Convention topics.

In This Section

[Cdecl, _cdecl, __cdecl](#)

[Pascal, _pascal, __pascal](#)

[_stdcall, __stdcall](#)

[_fastcall, __fastcall](#)

Cdecl, _cdecl, __cdecl

Category

Modifiers, Keyword extensions

Syntax

```
cdecl <data/function definition> ;_cdecl__cdecl
_cdecl <data/function definition> ;
__cdecl <data/function definition> ;
```

Description

Use a cdecl, _cdecl, or __cdecl modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use cdecl, _cdecl, or __cdecl in front of a function, it affects how the parameters are passed (parameters are pushed right to left, and the caller cleans up the stack). The __cdecl modifier overrides the compiler directives and IDE options.

The cdecl, _cdecl, and __cdecl keywords are specific to Borland C++.

Pascal, _pascal, __pascal

Category

Modifiers, Keyword extensions

Syntax

```
pascal <data-definition/function-definition> ;_pascal__pascal
_pascal <data-definition/function-definition> ;
__pascal <data-definition/function-definition> ;
```

Description

Use the pascal, _pascal, and __pascal keywords to declare a variable or a function using a Pascal-style naming convention (the name is in uppercase).

In addition, pascal declares Delphi language-style parameter-passing conventions when applied to a function header (parameters pushed left to right; the called function cleans up the stack).

In C++ programs, functions declared with the pascal modifier will still be mangled.

_stdcall, __stdcall

Category

Modifiers, Keyword extensions

Syntax

```
__stdcall <function-name> __stdcall  
__stdcall <function-name>
```

Description

The `__stdcall` and `__stdcall` keywords force the compiler to generate function calls using the Standard calling convention. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments.

Such functions comply with the standard WIN32 argument-passing convention.

Note: Note: The `__stdcall` modifier is subject to name mangling. See the description of the `-VC` option.

`__fastcall`, `__fastcall`

Category

Modifiers, Keyword extensions

Syntax

```
return-value __fastcall function-name(parm-list) __fastcall  
return-value __fastcall function-name(parm-list)
```

Description

Use the `__fastcall` modifier to declare functions that expect parameters to be passed in registers. The first three parameters are passed (from left to right) in EAX, EDX, and ECX, if they fit in the register. The registers are not used if the parameter is a floating-point or struct type.

All form class member functions must use the `__fastcall` convention.

The compiler treats this calling convention as a new language specifier, along the lines of `_cdecl` and `_pascal`.

Functions declared using `_cdecl` or `_pascal` cannot have the `__fastcall` modifier because they use the stack to pass parameters. Likewise, the `__fastcall` modifier cannot be used together with `_export`.

The compiler prefixes the `__fastcall` function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

For Microsoft C++ style `__fastcall` implementation, see `__msfastcall` and `__msreturn`.

Note: Note: The `__fastcall` modifier is subject to name mangling. See the description of the `-VC` option.

Multithread Variables

This section contains Multithread Variable topics.

In This Section

[__thread, Multithread Variables](#)

[__thread, Multithread Variables](#)

Category

Keyword extensions

Description

The keyword `__thread` is used in multithread programs to preserve a unique copy of global and static class variables. Each program thread maintains a private copy of a `__thread` variable for each thread.

The syntax is Type `__thread` variable`__name`. For example

```
int __thread x;
```

This statement declares an integer type variable that will be global but private to each thread in the program in which the statement occurs.

Function Modifiers

This section contains Function Modifier topics.

In This Section

[Function Modifiers](#)

Function Modifiers

This section presents descriptions of the function modifiers available with the Borland C++ compiler.

You can use the `__declspec(dllexport)`, and `__declspec(dllimport)` and `__saveregs` modifiers to modify functions.

In 32-bit programs the keyword can be applied to class, function, and variable declarations

The `__declspec(dllexport)` modifier makes the function exportable from Windows. The `__declspec(dllimport)` modifier makes a function available to a Windows program. The keywords are used in an executable (if you don't use smart callbacks) or in a DLL.

Functions declared with the `__fastcall` modifier have different names than their non-`__fastcall` counterparts. The compiler prefixes the `__fastcall` function name with an `@`. This prefix applies to both unmangled C function names and to mangled C++ function names.

Borland C++ modifiers

Modifier	Use with	Description
<code>const1</code>	Variables	Prevents changes to object.
<code>volatile1</code>	Variables	Prevents register allocation and some optimization. Warns compiler that object might be subject to outside change during evaluation.
<code>__cdecl2</code>	Functions	Forces C argument-passing convention. Affects linker and link-time names.
<code>__cdecl2</code>	Variables	Forces global identifier case-sensitivity and leading underscores in C.
<code>__pascal</code>	Functions	Forces Pascal argument-passing convention. Affects linker and link-time names.
<code>__pascal</code>	Variables	Forces global identifier case-insensitivity with no leading underscores in C.
<code>__import</code>	Functions/classes	Tells the compiler which functions or classes to import.
<code>__export</code>	Functions/classes	Tells the compiler which functions or classes to export.
<code>__declspec(dllimport)</code>	Functions/classes	Tells the compiler which functions or classes to import. This is the preferred method.
<code>__declspec(dllexport)</code>	Functions/classes	Tells the compiler which functions or classes to export. This is the preferred method.
<code>__fastcall</code>	Functions	Forces register parameter passing convention. Affects the linker and link-time names.
<code>__stdcall</code>	Function	Forces the standard WIN32 argument-passing convention.

1. C++ extends `const` and `volatile` to include classes and member functions.
2. This is the default.

Pointers

This section contains Pointer topics.

In This Section

[Pointers](#)

[Pointers To Objects](#)

[Pointers To Functions](#)

[Pointer Declarations](#)

[Pointer Constants](#)

[Pointer Arithmetic](#)

[Pointer Conversions](#)

[C++ Reference Declarations](#)

Pointers

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer categories have distinct properties, purposes, and rules for manipulation, although they do share certain characteristics. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers are numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Note: See Referencing for a discussion of referencing and dereferencing.

Pointers To Objects

A pointer of type "pointer to object of type" holds the address of (that is, points to) an object of type. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

Pointers To Functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called.

A pointer to a function has a type called "pointer to function returning type," where type is the function's return type. For example,

```
void (*func) ();
```

In C++, this is a pointer to a function taking no arguments, and returning void. In C, it's a pointer to a function taking an unspecified number of arguments and returning void. In this example,

```
void (*func)(int);
```

*func is a pointer to a function taking an int argument and returning void.

For C++, such a pointer can be used to access static member functions. Pointers to class members must use pointer-to-member operators. See `static_cast` for details.

Pointer Declarations

A pointer must be declared as pointing to some particular type, even if that type is void (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. The compiler lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to void. In C, but not C++, you can assign a void* pointer to a non-void* pointer. See `void` for details.

Warning: You need to initialize pointers before using them.

If type is any predefined or user-defined type, including void, the declaration

```
type *ptr;    /* Uninitialized pointer */
```

declares ptr to be of type "pointer to type." All the scoping, duration, and visibility rules apply to the ptr object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic NULL (defined in the standard library header files, such as `stdio.h`) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.

The pointer type "pointer to void" must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that vptr is a generic pointer capable of being assigned to by any "pointer to type" value, including null, without complaint. Assignments without proper casting between a "pointer to type1" and a "pointer to type2," where type1 and type2 are different types, can invoke a compiler warning or error. If type1 is a function and type2 isn't (or vice versa), pointer assignments are illegal. If type1 is a pointer to void, no cast is needed. Under C, if type2 is a pointer to void, no cast is needed.

Pointer Constants

A pointer or the pointed-at object can be declared with the `const` modifier. Anything declared as a `const` cannot be have its value changed. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```
int i;                // i is an int
int * pi;             // pi is a pointer to int (uninitialized)
int * const cp = &i;  // cp is a constant pointer to int
const int ci = 7;     // ci is a constant int
const int * pci;      // pci is a pointer to constant int
const int * const cpc = &ci; // cpc is a constant pointer to a
                          // constant int
```

The following assignments are legal:

```
i = ci;               // Assign const-int to int
*cp = ci;             // Assign const-int to
```

```

// object-pointed-at-by-a-const-pointer
++pci;           // Increment a pointer-to-const
pci = cpc;       // Assign a const-pointer-to-a-const to a
// pointer-to-const

```

The following assignments are illegal:

```

ci = 0;          // NO--cannot assign to a const-int
ci--;           // NO--cannot change a const-int
*pci = 3;        // NO--cannot assign to an object
// pointed at by pointer-to-const
cp = &ci;        // NO--cannot assign to a const-pointer,
// even if value would be unchanged
cpc++;          // NO--cannot change const-pointer
pi = pci;        // NO--if this assignment were allowed,
// you would be able to assign to *pci
// (a const value) by assigning to *pi.

```

Similar rules apply to the volatile modifier. Note that const and volatile can both appear as modifiers to the same identifier.

Pointer Arithmetic

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type "pointer to type" automatically take into account the size of type; that is, the number of bytes needed to store a type object.

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to type, adding an integral value to the pointer advances the pointer by that number of objects of type. If type has size 10 bytes, then adding an integer 5 to a pointer to type advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if ptr1 points to the third element of an array, and ptr2 points to the tenth element, then the result of ptr2 - ptr1 would be 7.

The difference between two pointers has meaning only if both pointers point into the same array

When an integral value is added to or subtracted from a "pointer to type," the result is also of type "pointer to type."

There is no such element as "one past the last element," of course, but a pointer is allowed to assume such a value. If P points to the last array element, P + 1 is legal, but P + 2 is undefined. If P points to one past the last array element, P - 1 is legal, giving a pointer to the last element. However, applying the indirection operator * to a "pointer to one past the last element" leads to undefined behavior.

Informally, you can think of P + n as advancing the pointer by (n * sizeof(type)) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type ptrdiff_t defined in stddef.h. This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of ptrdiff_t. In the expression P1 - P2, where P1 and P2 are of type pointer to type (or pointer to qualified type), P1 and P2 must point to existing elements or to one past the last element. If P1 points to the i-th element, and P2 points to the j-th element, P1 - P2 has the value (i - j).

Pointer Conversions

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (type*) will convert a pointer to type "pointer to type."

See C++ specific for a discussion of C++ typecast mechanisms.

C++ Reference Declarations

C++ reference types are closely related to pointer types. Reference types create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by value. In C++ you can pass arguments by value or by reference. See Referencing for complete details.

Arrays

The declaration

type declarator [<constant-expression>]

declares an array composed of elements of type. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. The following example shows one way to declare a two-dimensional array. The implementation is for three rows and five columns but it can be very easily modified to accept run-time user input.

```
/* DYNAMIC MEMORY ALLOCATION FOR A MULTIDIMENSIONAL OBJECT. */
#include <stdio.h>
#include <stdlib.h>
```

```
typedef long double TYPE;
typedef TYPE *OBJECT;
unsigned int rows = 3, columns = 5;
```

```
void de_allocate(OBJECT);
```

```
int main(void) {
    OBJECT matrix;
    unsigned int i, j;
```

```
/* STEP 1: SET UP THE ROWS. */
matrix = (OBJECT) calloc( rows, sizeof(TYPE *));
```

```
/* STEP 2: SET UP THE COLUMNS. */
for (i = 0; i < rows; ++i)
    matrix[i] = (TYPE *) calloc( columns, sizeof(TYPE));
```

```

    for (i = 0; i < rows; i++)
        for (j = 0; j < columns; j++)
            matrix[i][j] = i + j;    /* INITIALIZE */

```

```

for (i = 0; i < rows; ++i) {
    printf("\n\n");
    for (j = 0; j < columns; ++j)
        printf("%5.2Lf", matrix[i][j]);
    de_allocate(matrix);
    return 0;
}

```

```

void de_allocate(OBJECT x) {
    int i;

```

```

    for (i = 0; i < rows; i++)    /* STEP 1: DELETE THE COLUMNS */
        free(x[i]);

```

```

    free(x);    /* STEP 2: DELETE THE ROWS. */
}

```

This code produces the following output:

```

0.00 1.00 2.00 3.00 4.00
1.00 2.00 3.00 4.00 5.00
2.00 3.00 4.00 5.00 6.00

```

In certain contexts, the first array declarator of a series might have no expression inside the brackets. Such an array is of indeterminate size. This is legitimate in contexts where the size of the array is not needed to reserve space.

For example, an extern declaration of an array object does not need the exact dimension of the array; neither does an array function parameter. As a special extension to ANSI C, Borland C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a sizeof or & operator, an array type expression is converted to a pointer to the first element of the array.

Functions

This section contains Function topics.

In This Section

[Functions](#)

[Declarations And Definitions](#)

[Declarations And Prototypes](#)

[Definitions](#)

[Formal Parameter Declarations](#)

[Function Calls And Argument Conversions](#)

Functions

Functions are central to C and C++ programming. Languages such as Delphi distinguish between procedure and function. For C and C++, functions play both roles.

Member functions are sometimes referred to as methods.

Declarations And Definitions

Each program must have a single external function named `main` or `WinMain` marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are external by default and are normally accessible from any file in the program. You can restrict visibility of functions by using the static storage class specifier (see [Linkage](#)).

Functions are defined in your source files or made available by linking precompiled libraries.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide the compiler with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Note: In C++ you must always use function prototypes. We recommend that you also use them in C.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a definition and a declaration is that the definition has a function body.

Declarations And Prototypes

In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows

```
<type> func()
```

where `type` is the optional return type defaulting to `int`. In C++, this declaration means `<type> func(void)`. A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

This problem was eased by the introduction of function prototypes with the following declaration syntax:

```
<type> func(parameter-declarator-list);
```

Note: You can enable a warning within the IDE or with the command-line compiler: "Function called without a prototype."

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */
foo()
{
    int limit = 32;
    char ch = 'A';
    long mval;
    mval = lmax(limit,ch);    /* function call */
}
```

Since it has the function prototype for lmax, this program converts limit and ch to long, using the standard rules of assignment, before it places them on the stack for the call to lmax. Without the function prototype, limit and ch would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to lmax would not match in size or content what lmax was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function strcpy takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word void indicates a function that takes no arguments at all:

```
func(void);
```

In C++, func() also declares a function taking no arguments

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as printf), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Note: stdarg.h and varargs.h contain macros that you can use in user-defined functions with variable numbers of parameters.

Here are some more examples of function declarators and prototypes:

```
int f();/* In C, a function returning an int with no information about parameters.
This is the K&R "classic style." */
int f();/* In C++, a function taking no arguments */
int f(void);/* A function returning an int that takes no parameters. */
int p(int,long);/* A function returning an int that accepts two parameters: the first, an
int; the second, a long. */
int __pascal q(void);/* A pascal function returning an int that takes no parameters at all.
*/
```



```
int printf(char *format,...; /*A function returning an int and accepting a pointer to a
char fixedparameter and any number of additionalparameters of unknown type. */
int (*fp)(int)/* A pointer to a function returning an intand requiring an int parameter. */
```

Definitions

The general syntax for external function definitions is as follows:

External function definitions

```
file
    external-definition
    file external-definition
external-definition:
    function-definition
    declaration
    asm-statement
function-definition:
    <declaration-specifiers> declarator <declaration-list>
    compound-statement
```

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):

- 1. Optional storage class specifiers: `extern` or `static`. The default is `extern`.
- 2. A return type, possibly `void`. The default is `int`.
- 3. Optional modifiers: `__pascal`, `__cdecl`, `__export` `__saveregs`. The defaults depend on the compiler option settings.
- 4. The name of the function.
- 5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is `func(void)`. The old style of `func` is legal in C but antiquated and possibly unsafe.
- 6. A function body representing the code to be executed when the function is called.

Note: You can mix elements from 1 and 2.

Formal Parameter Declarations

The formal parameter declaration list follows a syntax similar to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) {                // no args
int func(T1 t1, T2 t2, T3 t3=1) { // three simple parameters, one
                                // with default argument
int func(T1* ptr1, T2& tref) {   // A pointer and a reference arg
int func(register int i) {       // Request register for arg
int func(char *str,...) {        /* One string arg with a variable number of other
                                args, or with a fixed number of args with varying types */
```

In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, or enumerations; pointers or references to structures and unions; or pointers to functions, classes, or arrays.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all have automatic scope and duration for the duration of the function. The only legal storage class specifier is register.

The const and volatile modifiers can be used with formal parameter declarators

Function Calls And Argument Conversions

A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters.

Here is a summary of the rules governing how the compiler deals with language modifiers and formal parameters in function calls, both with and without prototypes:

- The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
- A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, the compiler converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard arithmetic conversions. When a function prototype is in scope, the compiler converts the given argument to the type of the declared parameter as if by assignment

When a function prototype includes an ellipsis (...), the compiler converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need to be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If your function prototype does not match the actual function definition, the compiler will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Structures

This section contains Structure topics.

In This Section

[Structures](#)
[Untagged Structures And Typedefs](#)
[Structure Member Declarations](#)
[Structures And Functions](#)
[Structure Member Access](#)
[Structure Name Spaces](#)
[Incomplete Declarations](#)
[Bit Fields](#)

Structures

A structure is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The structure type lets you handle complex data structures almost as easily as single variables. Structure initialization is discussed in Arrays, structures, and unions.

In C++, a structure type is treated as a class type with certain differences: default access is public, and the default for the base class is also public. This allows more sophisticated control over access to structure members by using the C++ access specifiers: public (the default), private, and protected. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword struct. For example

```
struct mystruct { ... }; // mystruct is the structure tag
.
.
.
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

Untagged Structures And Typedefs

If you omit the structure tag, you can get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited struct-id-list to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a typedef while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];          // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT;    // no tag
YRSTRUCT y, *yp, arry[20];
```

Usually, you don't need both a tag and a typedef: either can be used in structure declarations.

Untagged structure and union members are ignored during initialization.

Structure Member Declarations

The member-decl-list within the braces declares the types and names of the structure members using the declarator syntax shown in Borland C++ declaration syntax.

A structure member can be of any type, with two exceptions

The member type cannot be the same as the struct type being currently declared:

```
struct mystruct { mystruct s } s1, s2; // illegal
```

However, a member can be a pointer to the structure being declared, as in the following example:

```
struct mystruct { mystruct *ps } s1, s2; // OK
```

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

Except in C++, a member cannot have the type "function returning...", but the type "pointer to function returning..." is allowed. In C++, a struct can have member functions.

Note: You can omit the struct keyword in C++.

Structures And Functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void); // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s);           // directly
void func2(mystruct *sptr);       // via a pointer
void func3(mystruct &sref);       // as a reference (C++ only)
```

Structure Member Access

Structure and union members are accessed using the following two selection operators:

- . (period)
- -> (right arrow)

Suppose that the object `s` is of struct type `S`, and `sptr` is a pointer to `S`. Then if `m` is a member identifier of type `M` declared in `S`, the expressions `s.m` and `sptr->m` are of type `M`, and both represent the member object `m` in `S`. The expression `sptr->m` is a convenient synonym for `(*sptr).m`.

The operator `.` is called the direct member selector and the operator `->` is called the indirect (or pointer) member selector. For example:

```
struct mystruct
{
    int i;
    char str[21];
    double d;
} s, *sptr = &s;
.
.
.
s.i = 3;           // assign to the i member of mystruct s
sptr -> d = 1.23;  // assign to the d member of mystruct s
```

The expression `s.m` is an lvalue, provided that `s` is an lvalue and `m` is not an array type. The expression `sptr->m` is an lvalue unless `m` is an array type.

If structure `B` contains a field whose type is structure `A`, the members of `A` can be accessed by two applications of the member selectors

```
struct A {
    int j;
    double x;
};
struct B {
    int i;
    struct A a;
    double d;
} s, *sptr;
.
.
.
s.i = 3;           // assign to the i member of B
s.a.j = 2;         // assign to the j member of A
sptr->d = 1.23;     // assign to the d member of B
sptr->a.x = 3.14    // assign to x member of A
```

Each structure declaration introduces a unique structure type, so that in

```
struct A {
    int i,j;
    double d;
} a, a1;
struct B {
    int i,j;
    double d;
} b;
```

the objects `a` and `a1` are both of type `struct A`, but the objects `a` and `b` are of different structure types. Structures can be assigned only if the source and destination have the same type:

```

a = a1;    // OK: same type, so member by member assignment
a = b;     // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign member-by-member */

```

Structure Name Spaces

Structure tag names share the same name space with union tags and enumeration tags (but enums within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, typedef names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example

```

goto s;
.
.
.
s:      // Label
struct s { // OK: tag and label name spaces different
    int s; // OK: label, tag and member name spaces different
    float s; // ILLEGAL: member name duplicated
} s;      // OK: var name space different. In C++, this can only
          // be done if s does not have a constructor.
union s { // ILLEGAL: tag space duplicate
    int s; // OK: new member space
    float f;
} f;      // OK: var name space
struct t {
    int s; // OK: different member space
    .
    .
    .
} s;      // ILLEGAL: var name duplicate

```

Incomplete Declarations

Incomplete declarations are also known as forward declarations.

A pointer to a structure type A can legally appear in the declaration of another structure B before A has been declared:

```

struct A; // incomplete
struct B { struct A *pa };
struct A { struct B *pb };

```

The first appearance of A is called incomplete because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of B doesn't need the size of A.

Bit Fields

Bit fields are specified numbers of bits that may or may not have an associated identifier. Bit fields offer a way of subdividing structures (structs, unions, classes) into named parts of user-defined sizes.

Declaring bit fields

You specify the bit-field width and optional identifier as follows:

```
type-specifier <bitfield-id> : width;
```

In C++, type-specifier is bool, char, unsigned char, short, unsigned short, long, unsigned long, int, unsigned int, __int64 or unsigned __int64. In strict ANSI C, type-specifier is int or unsigned int.

The expression width must be present and must evaluate to a constant integer. In C++, the width of a bit field may be declared of any size. In strict ANSI C, the width of a bit field may be declared only up to the size of the declared type. A zero length bit field skips to the next allocation unit.

If the bit field identifier is omitted, the number of bits specified in width is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused.

Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (. and ->) used for non-bit-field members.

Limitations of using bit fields

When using bit fields, be aware of the following issues:

- The code will be non-portable since the organization of bits-within-bytes and bytes-within-words is machine dependent.
- You cannot take the address of a bit field; so the expression &mystruct.x is illegal if x is a bit field identifier, because there is no guarantee that mystruct.x lies at a byte address.
- Bit fields are used to pack more variables into a smaller data space, but causes the compiler to generate additional code to manipulate these variables. This costs in terms of code size and execution time.

Because of these disadvantages, using bit fields is generally discouraged, except for certain low-level programming. A recommended alternative to having one-bit variables, or flags, is to use defines. For example:

```
#define Nothing 0x00
#define bitOne 0x01
#define bitTwo 0x02
#define bitThree 0x04
#define bitFour 0x08
#define bitFive 0x10
#define bitSix 0x20
#define bitSeven 0x40
#define bitEight 0x80
```

can be used to write code like:

```
if (flags & bitOne) {...}    // is bit One turned on
flags |= bitTwo;             // turn bit Two on
flags &= ~bitThree;          // turn bit Three off
```

Similar schemes can be made for bit fields of any size.

Padding of bit fields

In C++, if the width size is larger than the type of the bit field, the compiler will insert padding equal to the requested width size minus the size of the type of the bit field. So, declaration:

```
struct mystruct
{
    int i : 40;
    int j : 8;
};
```

will create a 32 bit storage for 'i', an 8 bit padding, and 8 bit storage for 'j'. To optimize access, the compiler will consider 'i' to be a regular int variable, not a bit field.

Layout and alignment

Bit fields are broken up into groups of consecutive bit fields of the same type, without regard to signedness. Each group of bit fields is aligned to the current alignment of the type of the members of the group. This alignment is determined by the type AND by the setting of the overall alignment (set by the byte alignment option `-aN`). Within each group, the compiler will pack the bit fields inside of areas as large as the type size of the bit fields. However, no bit field is allowed to straddle the boundary between 2 of those areas. The size of the total structure will be aligned, as determined by the current alignment.

Example of bit field layout, padding, and alignment

In the following C++ declaration, `my_struct` contains 6 bit fields of 3 different types, int, long, and char:

```
struct my_struct
{
    int one : 8;
    unsigned int two : 16;
    unsigned long three : 8;
    long four : 16;
    long five : 16;
    char six : 4;
};
```

Bit fields 'one' and 'two' will be packed into one 32-bit area.

Next, the compiler inserts padding, if necessary, based on the current alignment, and the type of three, because the type changes between the declarations of variables two and three. For example, if the current alignment is byte alignment (`-a1`), no padding is needed; whereas, if the alignment is 4 bytes (`-a4`), then 8-bit padding is inserted.

Next, variables three, four, and five are all of type long. Variables three and four are packed into one 32 bit area, but five can not be packed into that same area, since that would create an area of 40 bits, which is more than the 32 bit allowed for the long type. To start a new area for five, the compiler would insert no padding if the current alignment is byte alignment, or would insert 8 bits of padding if the current alignment is dword (4 byte) alignment.

With variable six, the type changes again. Since char is always byte aligned, no padding is needed. To force alignment for the whole struct, the compiler will finish up the last area with 4 bits of padding if byte alignment is used, or 12 bits of padding if dword alignment is used.

The total size of `my_struct` is 9 bytes with byte alignment, or 12 bytes with dword alignment.

To get the best results when using bit fields you should:

- Sort bit fields by type
- Make sure they are packed inside the areas by ordering them such that no bit field will straddle an area boundary
- Make sure the struct is filled as much as possible.

Another recommendation is to force byte alignment for this struct, by emitting `"#pragma option -a1"`. If you want to know how big your struct is, follow it by `"#pragma sizeof(mystruct)"`, which gives you the size.

Using one bit signed fields

For a signed type of one bit, the possible values are 0 or -1 . For an unsigned type of one bit, the possible values are 0 or 1. Note that if you assign "1" to a signed bit field, the value will be evaluated as -1 (negative one).

When storing the values true and false into a one bit sized bit field of a signed type, you can not test for equality to true because signed one bit sized bit fields can only hold the values '0' and '-1', which are not compatible with true and false. You can, however, test for non-zero.

For unsigned varieties of all types, and of course for the bool type, testing for equality to true will work as expected.

Thus:

```
struct mystruct
{
    int flag : 1;
} M;
int testing()
{
    M.flag = true;
    if(M.flag == true)
        printf("success");}
```

will NOT work, but:

```
struct mystruct
{
    int flag : 1;
} M;
int testing()
{
    M.flag = true;
    if(M.flag)
        printf("success");
}
```

works just fine.

Notes on compatibility

Between versions of the compiler, changes can be made to default alignment, or for purposes of compatibility with other compilers. Consequently, this could change the alignment of bit fields. Therefore, there is no guarantee that the alignment of bit fields will be consistent between versions of the compiler. To check the backward compatibility of bit fields in your code you can add an assert statement that checks for the structure size you are expecting.

According to the C and C++ language specifications, the alignment and storage of bit fields is implementation defined. Therefore, compilers can align and store bit fields differently. If you want complete control over the layout of bit fields, it is advisable to write your own bit field accessing routines and create your own bit fields.

Unions

This section contains Union topics.

In This Section

[Unions](#)

[Anonymous Unions](#)

[Union Declarations](#)

Unions

Union types are derived types sharing many of the syntactic and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
    int i;
    double d;
    char ch;
} mu, *muptr=&mu;
```

the identifier `mu`, of type `union myunion`, can be used to hold an `int`, an 8-byte `double`, or a single-byte `char`, but only one of these at the same time

Note: Unions correspond to the variant record types of Delphi.

`sizeof(union myunion)` and `sizeof(mu)` both return 8, but 4 bytes are unused (padded) when `mu` holds an `int` object, and 7 bytes are unused when `mu` holds a `char`. You access union members with the structure member selectors (`.` and `->`), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); //OK: displays mu.d = 4.016
printf("mu.i = %d\n",mu.i); //peculiar result
mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch); //OK: displays mu.ch = A
printf("mu.d = %f\n",mu.d); //peculiar result
muptr->i = 3;
printf("mu.i = %d\n",mu.i); //OK: displays mu.i = 3
```

The second `printf` is legal, since `mu.i` is an integer type. However, the bit pattern in `mu.i` corresponds to parts of the `double` previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Anonymous Unions

A union that does not have a tag and is not used to declare a named object (or other type) is called an anonymous union. It has the following form:

```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the `x.y` or `p->y` syntax.

Anonymous unions can be global, nested, or unnested. ANSI-C never allows anonymous unions. ANSI-C++ allows all three types of anonymous unions.

C++ Anonymous unions

An anonymous union cannot have member functions or private or protected members. At file level an anonymous union must be declared static. Local anonymous unions must be either automatic or static; in other words, an anonymous union cannot have external linkage. Unnested anonymous unions are only allowed in C++.

Nested anonymous unions

The outer structure, class, or union of a nested anonymous union must have a tag. Borland C and C++ allow nested anonymous unions by default. In C only, a nested anonymous union can, optionally, have a tag.

Example:

```
struct outer {  
    int x;  
};
```

```
int main(void)  
{  
    struct outer o;  
}
```

Union Declarations

The general declaration syntax for unions is similar to that for structures. The differences are:

- Unions can contain bit fields, but only one can be active. They all start at the beginning of the union. (Note that, because bit fields are machine dependent, they can pose problems when writing portable code.)
- Unlike C++ structures, C++ union types cannot use the class access specifiers: public, private, and protected. All fields of a union are public.
- Unions can be initialized only through their first declared member:

```
union local87 {  
    int i;  
    double d;  
} a = { 20 };
```

A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union can have a constructor.

Enumerations

This section contains Enumeration topics.

In This Section

[Enumerations](#)

[Assignment To Enum Types](#)

Enumerations

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration:

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, enum days, a variable anyday of this type, and a set of enumerators (sun, mon,...) with constant integer values.

The -b compiler switch controls the “Treat Enums As Ints” option. When this switch is used, the compiler allocates a whole word (a four-byte int) for enumeration types (variables of type enum). The default is ON (meaning enums are always ints) if the range of values permits, but the value is always promoted to an int when used in expressions. The identifiers used in an enumerator list are implicitly of type signed char, unsigned char, or int, depending on the values of the enumerators. If all values can be represented in a signed or unsigned char, then that is the type of each enumerator.

In C, a variable of an enumerated type can be assigned any value of type int--no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is:

```
anyday = mon;           // OK
anyday = 1;             // illegal, even though mon == 1
```

The identifier days is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type enum days:

```
enum days payday, holiday; // declare two variables
```

In C++, you can omit the enum keyword if days is not the name of anything else in the same scope.

As with struct and union declarations, you can omit the tag if no further variables of this enum type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

The enumerators listed inside the braces are also known as enumeration constants. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (sun) is set to zero, and each succeeding enumerator is set to one more than its predecessor (mon = 1, tues = 2, and so on). See Enumeration constants for more on enumeration constants.

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration:

```
/* Initializer expression can include previously declared enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
            quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, nickel the value 5, and quarter the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever int types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;           // OK
*daysptr = anyday;     // OK
mon = tues;             // ILLEGAL: mon is a constant
```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int mon = 11;
{
    enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
    /* enumerator mon hides outer declaration of int mon */
    struct days { int i, j;}; // ILLEGAL: days duplicate tag
    double sat;              // ILLEGAL: redefinition of sat
}
mon = 12;                    // back in int mon scope
```

In C++, enumerators declared within a class are in the scope of that class.

In C++ it is possible to overload most operators for an enumeration. However, because the =, [], (), and -> operators must be overloaded as member functions, it is not possible to overload them for an enum. See the example on how to overload the postfix and prefix increment operators.

Assignment To Enum Types

The rules for expressions involving enum types have been made stricter. The compiler enforces these rules with error messages if the compiler switch -A is turned on (which means strict ANSI C++).

Assigning an integer to a variable of enum type results in an error:

```
enum color
{
    red, green, blue
};
int f()
{
    color c;
    c = 0;
    return c;
}
```

The same applies when passing an integer as a parameter to a function. Notice that the result type of the expression flag1|flag2 is int:

```
enum e
{
    flag1 = 0x01,
```

```
    flag2 = 0x02
};
void p(e);
void f()
{
    p(flag1|flag2);
}
```

To make the example compile, the expression `flag1|flag2` must be cast to the enum type: `e (flag1|flag2)`.

Expressions

This section contains Expression topics.

In This Section

[Expressions](#)

[Precedence Of Operators](#)

[Expressions And C++](#)

[Evaluation Order](#)

[Errors And Overflows](#)

Expressions

An expression is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Borland C++ expressions, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Note: BorlandC++ expressions shows how identifiers and operators are combined to form grammatically legal "phrases."

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands. The standard conversions are detailed in Methods used in standard arithmetic conversions. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by the compiler (see Evaluation order).

Expressions can produce an lvalue, an rvalue, or no value. Expressions might cause side effects whether they produce a value or not

The precedence and associativity of the operators are summarized in Associativity and precedence of Borland C++ operators. The grammar in Borland C++ expressions, completely defines the precedence and associativity of the operators.

Borland C++ expressions

```
primary-expression:
    literal
    this (C++ specific)
    :: identifier (C++ specific)
    :: operator-function-name (C++ specific)
    :: qualified-name (C++ specific)
    (expression)
    name

literal:
    integer-constant
    character-constant
    floating-constant
    string-literal

name:
    identifier
    operator-function-name (C++ specific)
    conversion-function-name (C++ specific)

~ class-name (C++ specific)
    qualified-name (C++ specific)
```

```

qualified-name: (C++ specific)
    qualified-class-name :: name
postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression (<expression-list>)
    simple-type-name (<expression-list>) (C++ specific)

postfix-expression . name
postfix-expression -> name
postfix-expression ++

postfix-expression --
    const_cast < type-id > ( expression ) (C++ specific)
    dynamic_cast < type-id > ( expression ) (C++ specific)
    reinterpret_cast < type-id > ( expression ) (C++ specific)
    static_cast < type-id > ( expression ) (C++ specific)
    typeid ( expression ) (C++ specific)
    typeid ( type-name ) (C++ specific)
expression-list:
    assignment-expression
    expression-list , assignment-expression
unary-expression:
    postfix-expression
    ++ unary-expression
    - - unary-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-name )
    allocation-expression (C++ specific)
    deallocation-expression (C++ specific)
unary-operator: one of & * + - !
allocation-expression: (C++ specific)
    <::> new <placement> new-type-name <initializer>
    <::> new <placement> (type-name) <initializer>
placement: (C++ specific)
    (expression-list)
new-type-name: (C++ specific)
    type-specifiers <new-declarator>
new-declarator: (C++ specific)
    ptr-operator <new-declarator>
    new-declarator [<expression>]
deallocation-expression: (C++ specific)
    <::> delete cast-expression
    <::> delete [] cast-expression
cast-expression:
    unary-expression
    ( type-name ) cast-expression

pm-expression:
    cast-expression
    pm-expression .* cast-expression (C++ specific)
    pm-expression ->* cast-expression (C++ specific)
multiplicative-expression:
    pm-expression
    multiplicative-expression * pm-expression
    multiplicative-expression / pm-expression
    multiplicative-expression % pm-expression
additive-expression:
    multiplicative-expression

```

```

    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression
shift-expression:
additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression
relational-expression:
    shift-expression
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression
equality-expression:
    relational-expression
equality expression == relational-expression
equality expression != relational-expression
AND-expression:
    equality-expression
    AND-expression & equality-expression
exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ AND-expression
inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression
logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && inclusive-OR-expression
logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression
conditional-expression:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expression
assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression
assignment-operator: one of

=      *=      /=      %=      +=      -=

<<     =>      >=      &=      ^=      |=
expression:
    assignment-expression
    expression , assignment-expression
constant-expression:
    conditional-expression

```

Precedence Of Operators

There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator in the following table is indicated by its order in the table. The first category (on the first line) has the highest precedence. Operators on the same line have equal precedence.

Operators	Associativity
()	left to right
[]	
->	
::	
.	
!	right to left
~	
+	
-	
++	
--	
&	
*	
sizeof	
new	
delete	
.*	left to right
->*	
*	left to right
/	
%	
+	left to right
-	
<<	left to right
>>	
<	left to right
<=	
>	
>=	
==	left to right
!=	
&	left to right
^	left to right
	left to right
&&	left to right
	right to left
?:	left to right
=	right to left

```

*=
/=
%=
+=
-=
&=
^=
|=
<<=
>>=
,

```

left to right

Expressions And C++

C++ allows the overloading of certain standard C operators, as explained in Overloading Operator Functions. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator `==` might be defined in class complex to test the equality of two complex numbers without changing its normal usage with non-class data types.

An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the C language rules for operators and conversions might not apply to expressions in C++.

Evaluation Order

The order in which the compiler evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. For example, consider the expression

```
i = v[i++]; // i is undefined
```

The value of `i` depends on whether `i` is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for `sum` and `total`. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value

The compiler regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression

Errors And Overflows

Associativity and precedence of Borland C++ operators. summarizes the precedence and associativity of the operators. During the evaluation of an expression, the compiler can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2^n arithmetic on n-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines. See `_matherr` and `signal`.

Operators Summary

This section contains Operator Summary topics.

In This Section

[Operators Summary](#)

Operators Summary

Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

Arithmetic

Assignment

Bitwise

C++ specific

Comma

Conditional

Equality

Logical

Postfix Expression Operators

Primary Expression Operators

Preprocessor

Reference/Indirect operators

Relational

sizeof

typeid

All operators can be overloaded except the following:

- . C++ direct component selector
- .* C++ dereference
- :: C++ scope access/resolution
- ?: Conditional

Depending on context, the same operator can have more than one meaning. For example, the ampersand (&) can be interpreted as:

- a bitwise AND (A & B)
- an address operator (&A)
- in C++, a reference modifier

Note: No spaces are allowed in compound operators. Spaces change the meaning of the operator and will generate an error.

Primary Expression Operators

This section contains Primary Expression Operator topics.

In This Section

[Primary Expression Operators](#)

Primary Expression Operators

For ANSI C, the primary expressions are literal (also sometimes referred to as constant), identifier, and (expression). The C++ language extends this list of primary expressions to include the keyword `this`, scope resolution operator `::`, name, and the class destructor `~` (tilde).

The primary expressions are summarized in the following list.

primary-expression:

- literal `this` (C++ specific)
- `::` identifier (C++ specific)
- `::` operator-function-name (C++ specific)
- `::` qualified-name (C++ specific)
- (expression) name

literal:

- integer-constant
- character-constant
- floating-constant
- string-literal

name:

- identifier
- operator-function-name (C++ specific)
- conversion-function-name (C++ specific)
- `~` class-name (C++ specific)
- qualified-name (C++ specific)

qualified-name: (C++ specific)

qualified-class-name `::` name

For a discussion of the primary expression `this`, see `this` (keyword). The keyword `this` cannot be used outside a class member function body.

The discussion of the scope resolution operator `::` begins on page 104. The scope resolution operator allows reference to a type, object, function, or enumerator even though its identifier is hidden.

The parenthesis surrounding an expression do not change the unadorned expression itself.

The primary expression name is restricted to the category of primary expressions that sometimes appear after the member access operators `.` (dot) and `->`. Therefore, name must be either an lvalue or a function. See also the discussion of member access operators.

An identifier is a primary expression, provided it has been suitably declared. The description and formal definition of identifiers is shown in Lexical Elements: Identifiers.

See the discussion on how to use the destructor operator `~` (tilde).

Postfix Expression Operators

This section contains Postfix Expression Operator topics.

In This Section

[Array Subscript Operator](#)

[Function Call Operator](#)

[. \(direct Member Selector\)](#)

[-> \(indirect Member Selector\)](#)

[Increment/decrement Operators](#)

Array Subscript Operator

Brackets ([]) indicate single and multidimensional array subscripts. The expression

```
<exp1>[exp2]
```

is defined as

```
*((exp1) + (exp2))
```

where either:

- exp1 is a pointer and exp2 is an integer or
- exp1 is an integer and exp2 is a pointer

Function Call Operator

Syntax

```
postfix-expression(<arg-expression-list>)
```

Remarks

Parentheses ()

- group expressions
- isolate conditional expressions
- indicate function calls and function parameters

The value of the function call expression, if it has a value, is determined by the return statement in the function definition.

This is a call to the function given by the postfix expression.

arg-expression-list is a comma-delimited list of expressions of any type representing the actual (or real) function arguments.

. (direct Member Selector)

Syntax

```
postfix-expression . identifier
```

postfix-expression must be of type union or structure.

identifier must be the name of a member of that structure or union type.

Remarks

Use the selection operator (.) to access structure and union members.

Suppose that the object *s* is of struct type *S* and *sptr* is a pointer to *S*. Then, if *m* is a member identifier of type *M* declared in *S*, this expression:

```
s.m
```

are of type *M*, and represent the member object *m* in *s*.

-> (indirect Member Selector)

Syntax

```
postfix-expression -> identifier
```

postfix-expression must be of type pointer to structure or pointer to union.

identifier must be the name of a member of that structure or union type.

The expression designates a member of a structure or union object. The value of the expression is the value of the selected member it will be an lvalue if and only if the postfix expression is an lvalue.

Remarks

You use the selection operator -> to access structure and union members.

Suppose that the object *s* is of struct type *S* and *sptr* is a pointer to *S*. Then, if *m* is a member identifier of type *M* declared in *S*, this expression:

```
sptr->m
```

is of type *M*, and represents the member object *m* in *s*.

The expression

```
s->sptr
```

is a convenient synonym for *(*sptr).m*.

Increment/decrement Operators

Increment operator (++)

Syntax

postfix-expression ++	(postincrement)
++ unary-expression	(preincrement)

The expression is called the operand. It must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue..

Postincrement operator

The value of the whole expression is the value of the postfix expression before the increment is applied.

After the postfix expression is evaluated, the operand is incremented by 1.

Preincrement operator

The operand is incremented by 1 before the expression is evaluated. The value of the whole expression is the incremented value of the operand.

The increment value is appropriate to the type of the operand.

Pointer types follow the rules for pointer arithmetic.

Decrement operator (--)

Syntax

```
postfix-expression --      (postdecrement)
-- unary-expression        (predecrement)
```

The decrement operator follows the same rules as the increment operator, except that the operand is decremented by 1 after or before the whole expression is evaluated.

Unary Operators

This section contains Unary Operator topics.

In This Section

[Unary Operators](#)
[Reference/deference Operators](#)
[Plus And Minus Operators](#)
[Arithmetic Operators](#)
[Sizeof](#)

Unary Operators

Syntax

```
<unary-operator> <unary expression>
```

OR

```
<unary-operator> <type><unary expression>
```

Remarks

Unary operators group right-to-left.

The C++ language provides the following unary operators:

- ! Logical negation
- * Indirection
- ++ Increment
- ~ Bitwise complement
- -- Decrement
- - Unary minus
- + Unary plus

Reference/deference Operators

Syntax

```
& cast-expression  
* cast-expression
```

Remarks

The & and * operators work together to reference and dereference pointers that are passed to functions.

Referencing operator (&)

Use the reference operator to pass the address of a pointer to a function outside of main().

The cast-expression operand must be one of the following:

- a function designator

- an lvalue designating an object that is not a bit field and is not declared with a register storage class specifier

If the operand is of type <type>, the result is of type pointer to <type>.

Some non-lvalue identifiers, such as function names and array names, are automatically converted into “pointer-to-X” types when they appear in certain contexts. The & operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following example:

```
T t1 = 1, t2 = 2;
T *ptr = &t1;      // Initialized pointer
*ptr = t2;         // Same effect as t1 = t2
```

T *ptr = &t1 is treated as

```
T *ptr;
ptr = &t1;
```

So it is ptr, or *ptr, that gets assigned. Once ptr has been initialized with the address &t1, it can be safely dereferenced to give the lvalue *ptr.

Indirection operator (*)

Use the asterisk (*) in a variable expression to create pointers. And use the indirect operator in external functions to get a pointer's value that was passed by reference.

If the operand is of type pointer to function, the result is a function designator.

If the operand is a pointer to an object, the result is an lvalue designating that object.

The result of indirection is undefined if either of the following occur:

- 1. The cast-expression is a null pointer.
- 2. The cast-expression is the address of an automatic variable and execution of its block has terminated.

Note: & can also be the bitwise AND operator.

Note: * can also be the multiplication operator.

Plus And Minus Operators

Unary

In these unary + - expressions

```
+ cast-expression
- cast-expression
```

the cast-expression operand must be of arithmetic type.

Results

+ cast-expression Value of the operand after any required integral promotions.

- cast-expression Negative of the value of the operand after any required integral promotions.

Binary

Syntax

```
add-expression + multiplicative-expression  
add-expression - multiplicative-expression
```

Legal operand types for op1 + op2:

- 1. Both op1 and op2 are of arithmetic type.
- 2. op1 is of integral type, and op2 is of pointer to object type.
- 3. op2 is of integral type, and op1 is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Legal operand types for op1 - op2:

- 1. Both op1 and op2 are of arithmetic type.
- 2. Both op1 and op2 are pointers to compatible object types.
- 3. op1 is of pointer to object type, and op2 is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Note: The unqualified type <type> is considered to be compatible with the qualified types const type, volatile type, and const volatile type.

Arithmetic Operators

Syntax

```
+ cast-expression  
- cast-expression  
add-expression + multiplicative-expression  
add-expression - multiplicative-expression  
multiplicative-expr * cast-expr  
multiplicative-expr / cast-expr  
multiplicative-expr % cast-expr  
postfix-expression ++      (postincrement)  
++ unary-expression        (preincrement)  
postfix-expression --      (postdecrement)  
-- unary-expression         (predecrement)
```

Remarks

Use the arithmetic operators to perform mathematical computations.

The unary expressions of + and - assign a positive or negative value to the cast-expression.

+ (addition), - (subtraction), * (multiplication), and / (division) perform their basic algebraic arithmetic on all data types, integer and floating point.

% (modulus operator) returns the remainder of integer division and cannot be used with floating points.

++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.

-- (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Sizeof

Category

Operators

Description

The sizeof operator has two distinct uses:

sizeof unary-expression

sizeof (type-name)

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type char (signed or unsigned), sizeof gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals sizeof array/ sizeof array[0] .

If the operand is a parameter declared as array type or function type, sizeof gives the size of the pointer. When applied to structures and unions, sizeof gives the total number of bytes, including any padding.

You cannot use sizeof with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of sizeof is size_t.

You can use sizeof in preprocessor directives; this is specific to Borland C++.

In C++, sizeof(classtype), where classtype is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Example

```
/* USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>
struct st {
    char *name;
    int age;
    double height;
};
struct st St_Array[] = { /* AN ARRAY OF structs */
    { "Jr.", 4, 34.20 }, /* St_Array[0] */
    { "Suzie", 23, 69.75 }, /* St_Array[1] */
};
int main()
{
    long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };
    printf("\nNumber of elements in LD_Array = %d",
        sizeof(LD_Array) / sizeof(LD_Array[0]));
    /**** THE NUMBER OF ELEMENTS IN THE St_Array. *****/
    printf("\nSt_Array has %d elements",
        sizeof(St_Array)/sizeof(St_Array[0]));
}
```

```
/***** THE NUMBER OF BYTES IN EACH St_Array ELEMENT. *****/
printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));
/***** THE TOTAL NUMBER OF BYTES IN St_Array. *****/
printf("\nSt_Array=%d", sizeof(St_Array));
return 0;
}
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 16
St_Array= 32
```


Binary Operators

This section contains Binary Operator topics.

In This Section

- [Binary Operators](#)
- [Multiplicative Operators](#)
- [Bitwise Operators](#)
- [Relational Operators](#)
- [Equality Operators](#)
- [Logical Operators](#)
- [Conditional Operators](#)
- [Assignment Operators](#)
- [Comma Operator](#)
- [C++ Specific Operators](#)

Binary Operators

These are the binary operators in Borland C++:

- Arithmetic + Binary plus (add)
- - Binary minus (subtract)
- * Multiply
- / Divide
- % Remainder (modulus)
- Bitwise << Shift left
- >> Shift right
- & Bitwise AND
- ^ Bitwise XOR (exclusive OR)
- | Bitwise inclusive OR
- Logical && Logical AND
- || Logical OR
- Assignment = Assignment
- *= Assign product
- /= Assign quotient
- %= Assign remainder (modulus)
- += Assign sum
- -= Assign difference
- <<= Assign left shift
- >>= Assign right shift
- &= Assign bitwise AND
- ^= Assign bitwise XOR
- |= Assign bitwise OR
- Relational < Less than
- > Greater than

- <= Less than or equal to
- >= Greater than or equal to
- == Equal to
- != Not equal to
- Component selection . Direct component selector
- -> Indirect component selector
- Class-member :: Scope access/resolution
- .* Dereference pointer to class member
- ->* Dereference pointer to class member
- Conditional ? : Actually a ternary operator for example,
- a ? x : y "if a then x else y"
- Comma , Evaluate

Multiplicative Operators

Syntax

```

multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr

```

Remarks

There are three multiplicative operators:

- * (multiplication)
- / (division)
- % (modulus or remainder)

The usual arithmetic conversions are made on the operands.

(op1 * op2) Product of the two operands

(op1 / op2) Quotient of (op1 divided by op2)

(op1 % op2) Remainder of (op1 divided by op2)

For / and %, op2 must be nonzero op2 = 0 results in an error. (You can't divide by zero.)

When op1 and op2 are integers and the quotient is not an integer:

- 1. If op1 and op2 have the same sign, op1 / op2 is the largest integer less than the true quotient, and op1 % op2 has the sign of op1.
- 2. If op1 and op2 have opposite signs, op1 / op2 is the smallest integer greater than the true quotient, and op1 % op2 has the sign of op1.

Note: Rounding is always toward zero.

* is context sensitive and can be used as the pointer reference operator.

Bitwise Operators

Syntax

```
AND-expression & equality-expression  
exclusive-OR-expr ^ AND-expression  
inclusive-OR-expr exclusive-OR-expression  
~cast-expression  
shift-expression << additive-expression  
shift-expression >> additive-expression
```

Remarks

Use the bitwise operators to modify the individual bits rather than the number.

Operator	What it does
&	bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
	bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
^	bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
~	bitwise complement; inverts each bit. ~ is used to create destructors.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends.
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.

Both operands in a bitwise expression must be of an integral type.

A	B	A & B	A ^ B	A B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Note: &, >>, << are context sensitive. & can also be the pointer reference operator.

Note: >> is often overloaded to be the input operator in I/O expressions. << is often overloaded to be the output operator in I/O expressions.

Relational Operators

Syntax

```
relational-expression < shift-expression  
relational-expression > shift-expression  
relational-expression <= shift-expression  
relational-expression >= shift-expression
```

Remarks

Use relational operators to test equality or inequality of expressions. If the statement evaluates to be true it returns a nonzero character; otherwise it returns false (0).

> greater than
< less than
>= greater than or equal
<= less than or equal

In the expression

```
E1 <operator> E2
```

the operands must follow one of these conditions:

- 1. Both E1 and E2 are of arithmetic type.
- 2. Both E1 and E2 are pointers to qualified or unqualified versions of compatible types.
- 3. One of E1 and E2 is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void.
- 4. One of E1 or E2 is a pointer and the other is a null pointer constant.

Equality Operators

There are two equality operators: == and !=. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

Note: Notice that == and != have a lower precedence than the relational operators < and >, <=, and >=. Also, == and != can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

```
equality-expression:==!=  
relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression
```

Logical Operators

Syntax

```
logical-AND-expr  && inclusive-OR-expression  
logical-OR-expr   || logical-AND-expression  
! cast-expression
```

Remarks

Operands in a logical expression must be of scalar type.

&& logical AND; returns true only if both expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to false, the second expression is not evaluated.

|| logical OR; returns true if either of the expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to true, the second expression is not evaluated.

! logical negation; returns true if the entire expression evaluates to be nonzero, otherwise returns false. The expression !E is equivalent to (0 == E).

Conditional Operators

Syntax

```
logical-OR-expr ? expr : conditional-expr
```

Remarks

The conditional operator ?: is a ternary operator.

In the expression E1 ? E2 : E3, E1 evaluates first. If its value is true, then E2 evaluates and E3 is ignored. If E1 evaluates to false, then E3 evaluates and E2 is ignored.

The result of E1 ? E2 : E3 will be the value of either E2 or E3 depending upon which evaluates.

E1 must be a scalar expression. E2 and E3 must obey one of the following rules:

- 1. Both of arithmetic type. E2 and E3 are subject to the usual arithmetic conversions, which determines the resulting type.
- 2. Both of compatible struct or union types. The resulting type is the structure or union type of E2 and E3.
- 3. Both of void type. The resulting type is void.
- 4. Both of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
- 5. One operand is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
- 6. One operand is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void. The resulting type is that of the non-pointer-to-void operand.

Assignment Operators

Syntax

```
unary-expr assignment-op assignment-expr
```

Remarks

The assignment operators are:

=	*=	/=	%=	+=	--=
<<=	>>=	&=	^=	=	

The = operator is the only simple assignment operator, the others are compound assignment operators.

In the expression E1 = E2, E1 must be a modifiable lvalue. The assignment expression itself is not an lvalue.

The expression

```
E1 op= E2
```

has the same effect as

```
E1 = E1 op E2
```

except the lvalue E1 is evaluated only once. For example, E1 += E2 is the same as E1 = E1 + E2.

The expression's value is E1 after the expression evaluates.

For both simple and compound assignment, the operands E1 and E2 must obey one of the following rules:

- 1. E1 is a qualified or unqualified arithmetic type and E2 is an arithmetic type.
- 2. E1 has a qualified or unqualified version of a structure or union type compatible with the type of E2.
- 3. E1 and E2 are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
- 4. Either E1 or E2 is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
- 5. E1 is a pointer and E2 is a null pointer constant.

Note: Spaces separating compound operators (+<space>=) will generate errors.

Note: There are certain conditions where assignment operators are not supported when used with properties.

Comma Operator

Syntax

```
expression , assignment-expression
```

Remarks

The comma separates elements in a function argument list.

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

The left operand E1 is evaluated as a void expression, then E2 is evaluated to give the result and type of the comma expression. By recursion, the expression

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each Ei, with the value and type of En giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls func with three arguments (i, 5, k), not four.

C++ Specific Operators

The operators specific to C++ are:

Operator	Meaning
::	Scope access (or resolution) operator

<code>.*</code>	Dereference pointers to class members
<code>->*</code>	Dereference pointers to pointers to class members
<code>const_cast</code>	adds or removes the <code>const</code> or <code>volatile</code> modifier from a type
<code>delete</code>	dynamically deallocates memory
<code>dynamic_cast</code>	converts a pointer to a desired type
<code>new</code>	dynamically allocates memory
<code>reinterpret_cast</code>	replaces casts for conversions that are unsafe or implementation dependent
<code>static_cast</code>	converts a pointer to a desired type
<code>typeid</code>	gets run-time identification of types and expressions

Use the scope access (or resolution) operator `::`(two semicolons) to access a global (or file duration) name even if it is hidden by a local redeclaration of that name.

Use the `.*` and `->*` operators to dereference pointers to class members and pointers to pointers to class members.

Statements

This section contains Statement topics.

In This Section

[Statements](#)

[Blocks](#)

[Labeled Statements](#)

[Expression Statements](#)

[Selection Statements](#)

[Iteration Statements](#)

[Jump Statements](#)

Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. Borland C++ statements shows the syntax for statements.

Borland C++ statements

statement:

labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

asm-statement

declaration (C++ specific)

labeled-statement:

identifier : statement

case constant-expression : statement

default : statement

compound-statement:

{ <declaration-list> <statement-list> }

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

expression-statement:

<expression> ;
 asm-statement:
 asm tokens newline
 asm tokens;
 asm { tokens; <tokens;>= <tokens;>}
 selection-statement:
 if (expression) statement
 if (expression) statement else statement
 switch (expression) statement
 iteration-statement:
 while (expression) statement
 do statement while (expression) ;
 for (for-init-statement <expression> ; <expression>) statement
 for-init-statement:
 expression-statement
 declaration (C++ specific)
 jump-statement:
 goto identifier ;
 continue ;
 break ;
 return <expression> ;

Blocks

A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces ({ }). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.

Labeled Statements

A statement can be labeled in two ways:

- label-identifier : statement

The label identifier serves as a target for the unconditional goto statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.

- case constant-expression : statement

default: statement

Case and default labeled statements are used only in conjunction with switch statements.

Expression Statements

Any expression followed by a semicolon forms an expression statement:

```
<expression>;
```

The compiler executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls

The null statement is a special case, consisting of a single semicolon (;). The null statement does nothing, and is therefore useful in situations where C++ syntax expects a statement but your program does not need one.

Selection Statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the if...else and the switch.

Iteration Statements

Iteration statements let you loop a set of statements. There are three forms of iteration in C++: while, do while, and for loops.

Jump Statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: break, continue, goto, and return

C++ Specifics

This section contains C++ Specific topics.

In This Section

- [C++ Specifics](#)
- [C++ namespaces](#)
- [New-style Typecasting Overview](#)
- [Run-time Type Identification \(RTTI\)](#)
- [Referencing](#)
- [The Scope Resolution Operator](#)
- [The new And delete Operators](#)
- [Classes](#)
- [Constructors And Destructors](#)
- [Operator Overloading Overview](#)
- [Overloading Operator Functions Overview](#)
- [Polymorphic Classes](#)
- [C++ Scope](#)
- [Templates](#)
- [Function Templates Overview](#)
- [Class Templates Overview](#)
- [Compiler Template Switches](#)
- [Template Generation Semantics](#)
- [Exporting And Importing Templates](#)

C++ Specifics

C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function `func` declared twice in C with different argument types causes a duplicated name error. Under C++, however, `func` will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. This topic discusses the aspects of C++ that can be used independently of classes, then describes the specifics of classes and class mechanisms.

See C++ Exception Handling and C-Based Structured Exceptions for details on compiling C and C++ programs with exception handling.

C++ namespaces

This section contains C++ namespace topics.

In This Section

- [Defining A namespace](#)
- [Declaring A namespace](#)
- [namespace Alias](#)
- [Extending A namespace](#)
- [Anonymous namespaces](#)
- [Accessing Elements Of A namespace](#)
- [Using Directive](#)
- [Explicit Access Qualification](#)

Defining A namespace

The grammar for defining a namespace is

```
original-namespace-name:  
    identifier  
namespace-definition:  
    original-namespace-definition  
    extension-namespace-definition  
    unnamed-namespace-definition
```

Grammatically, there are three ways to define a namespace with the namespace keyword:

```
original-namespace-definition:  
    namespace identifier { namespace-body }  
extension-namespace-definition:  
    namespace original-namespace-name { namespace-body }  
unnamed-namespace-definition:  
    namespace { namespace-body }
```

The body is an optional sequence of declarations. The grammar is

```
namespace-body:  
    declaration-seq opt
```

Declaring A namespace

An original namespace declaration should use an identifier that has not been previously used as a global identifier.

```
namespace alpha { /* ALPHA is the identifier of this namespace. */  
    /* your program declarations */  
    long double LD;  
    float f(float y) { return y; }  
}
```

A namespace identifier must be known in all translation units where you intend to access its elements.

namespace Alias

You can use an alternate name to refer to a namespace identifier. An alias is useful when you need to refer to a long, unwieldy namespace identifier.

```
namespace BORLAND_SOFTWARE_CORPORATION {
    /* namespace-body */
    namespace NESTED_BORLAND_SOFTWARE_CORPORATION {
        /* namespace-body */
    }
}
// Alias namespace
namespace BI = BORLAND_SOFTWARE_CORPORATION;
// Use access qualifier to alias a nested namespace
namespace NBI = BORLAND_SOFTWARE_CORPORATION::NESTED_BORLAND_SOFTWARE_CORPORATION;
```

Extending A namespace

Namespaces are discontinuous and open for additional development. If you redeclare a namespace, the effect is that you extend the original namespace by adding new declarations. Any extensions that are made to a namespace after a using-declaration, will not be known at the point at which the using-declaration occurs. Therefore, all overloaded versions of some function should be included in the namespace before you declare the function to be in use.

Anonymous namespaces

The C++ grammar allows you to define anonymous namespaces. To do this, you use the keyword namespace with no identifier before the enclosing brace.

```
namespace {           // Anonymous namespace
    // Declarations
}
```

All anonymous, unnamed namespaces in global scope (that is, unnamed namespaces that are not nested) of the same translation unit share the same namespace. This way you can make static declarations without using the static keyword.

Each identifier that is enclosed within an unnamed namespace is unique within the translation unit in which the unnamed namespace is defined.

Accessing Elements Of A namespace

There are three ways to access the elements of a namespace: by explicit access qualification, the using-declaration, or the using-directive. Remember that no matter which namespace you add to your local scope, identifiers in global scope (global scope is just another namespace) are still accessible by using the scope resolution operator ::.

- Explicit access qualification
- Using directive
- Using declaration

Accessing namespaces in classes

You cannot use a using directive inside a class. However, the using declarative is allowed and can be quite useful.

Using Directive

If you want to use several (or all of) the members of a namespace, C++ provides an easy way to get access to the complete namespace. The using-directive causes all identifiers in a namespace to be in scope at the point that the using-directive statement is made. The grammar for the using-directive is as follows.

using-directive:

using namespace :: opt nested-name-specifier opt namespace-name;

The using-directive is transitive. When you apply the using directive to a namespace that contains using directives within itself, you get access to those namespaces as well. For example, if you apply the using directive in your program, you also get namespaces `qux`, `foo`, and `bar`.

```
namespace qux {  
    using namespace foo; // This has been defined previously  
    using namespace bar; // This also has been defined previously  
}
```

The using-directive does not add any identifiers to your local scope. Therefore, an identifier defined in more than one namespace won't be a problem until you actually attempt to use it. Local scope declarations take precedence by hiding all other similar declarations.

Warning: Do not use the using directive in header files. You might accidentally break namespaces in client code.

Explicit Access Qualification

You can explicitly qualify each member of a namespace. To do so, you use the namespace identifier together with the `::` scope resolution operator followed by the member name. For example, to access a specific member of namespace ALPHA, you write:

```
ALPHA::LD; // Access a variable  
ALPHA::f;  // Access a function
```

Explicit access qualification can always be used to resolve ambiguity. No matter which namespace (except anonymous namespace) is being used in your subsystem, you can apply the scope resolution operator `::` to access identifiers in any namespace (including a namespace already being used in the local scope) or the global namespace. Therefore, any identifier in the application can be accessed with sufficient qualification.

New-style typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

New-style Typecasting Overview

This section contains New-style Typecasting Overview topics.

In This Section

[New-style Typecasting](#)

[const_cast \(typecast Operator\)](#)

[dynamic_cast \(typecast Operator\)](#)

[reinterpret_cast \(typecast Operator\)](#)

[static_cast \(typecast Operator\)](#)

New-style Typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

const_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
const_cast< T > (arg)
```

Description

Use the const_cast operator to add or remove the const or volatile modifier from a type.

In the statement, const_cast< T > (arg), T and arg must be of the same type except for const and volatile modifiers. The cast is resolved at compile time. The result is of type T. Any number of const or volatile modifiers can be added or removed with a single const_cast expression.

A pointer to const can be converted to a pointer to non-const that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A const object or a reference to const cast results in a non-const object or reference that is otherwise an identical type.

The const_cast operator performs similar typecasts on the volatile modifier. A pointer to volatile object can be cast to a pointer to non-volatile object without otherwise changing the type of the object. The result is a pointer to the original object. A volatile-type object or a reference to volatile-type can be converted into an identical non-volatile type.

dynamic_cast (typecast Operator)

Category

C++-Specific Keywords

Description

In the expression, dynamic_cast< T > (ptr), T must be a pointer or a reference to a defined class type or void*. The argument ptr must be an expression that resolves to a pointer or reference.

If T is void* then ptr must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if T is a pointer and ptr is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If T is a reference and ptr is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at runtime.

If successful, `dynamic_cast< T > (ptr)` converts ptr to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the `Bad_cast` exception is thrown.

Note: Runtime type identification (RTTI) is required for `dynamic_cast`.

reinterpret_cast (typeid Operator)

Category

C++-Specific Keywords

Syntax

```
reinterpret_cast< T > (arg)
```

Description

In the statement, `reinterpret_cast< T > (arg)`, T must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral arg can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

static_cast (typeid Operator)

Category

C++-Specific Keywords

Syntax

```
static_cast< T > (arg)
```

Description

In the statement, `static_cast< T > (arg)`, T must be a pointer, reference, arithmetic type, or enum type. Both T and arg must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using `static_cast` achieves exactly the same thing.

Integral types can be converted to enum types. A request to convert `arg` to a value that is not an element of enum is undefined.

The null pointer is converted to the null pointer value of the destination type, `T`.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class `X` to a pointer to some class `Y` if `X` is a base class for `Y`. A static conversion can be made only under the following conditions:

- if an unambiguous conversion exists from `Y` to `X`
- if `X` is not a virtual base class

An object can be explicitly converted to a reference type `X&` if a pointer to that object can be explicitly converted to an `X*`. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.

An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.

A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When `T` is a reference the result of `static_cast< T > (arg)` is an lvalue. The result of a pointer or reference cast refers to the original expression.

Run-time Type Identification (RTTI)

This section contains Run-time Type Identification (RTTI) topics.

In This Section

[Runtime Type Identification \(RTTI\) Overview](#)

[The typeid Operator](#)

Runtime Type Identification (RTTI) Overview

Runtime type identification (RTTI) lets you write portable code that can determine the actual type of a data object at runtime even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. Use the `dynamic_cast` operator to make runtime casts.

The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with `typeid` operator, which determines the actual type of its argument and returns a reference to an object of type `const type_info`, which describes that type.

You can also use a type name as the argument to `typeid`, and `typeid` will return a reference to a `const type_info` object for that type. The class `type_info` provides an `operator==` and an `operator!=` that you can use to determine whether two objects are of the same type. Class `type_info` also provides a member function `name` that returns a pointer to a character string that holds the name of the type.

The typeid Operator

This section contains typeid Operator topics.

In This Section

[__rtti, -RT Option](#)

[Runtime Type Identification And Destructors](#)

[__rtti, -RT Option](#)

Category (`__rtti` keyword)

Modifiers, C++ Keyword Extensions, C++-Specific Keywords

Description

Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the `-RT-` option to disable it or `-RT` to enable it.

If RTTI is disabled, or if the argument to `typeid` is a pointer or a reference to a non-polymorphic class, `typeid` returns a reference to a `const type_info` object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic runtime type identification (where appropriate) by using the `__rtti` keyword in the class definition.

When runtime type identification is disabled, if any base class is declared `__rtti`, then all polymorphic base classes must also be declared `__rtti`.

```
struct __rtti S1 { virtual s1func(); }; /* Polymorphic */
struct __rtti S2 { virtual s2func(); }; /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism, type information might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations without RTTI, you should declare `X` with the `__rtti` modifier. Otherwise, switching the order of the base classes for the class `X` results in the compile-time error "Can't inherit non-RTTI class from RTTI base 'S1'."

```
struct __rtti S1 { virtual func(); }; /* Polymorphic class */
struct S2 { }; /* Non-polymorphic class */
struct __rtti X : S1, S2 { };
```

Note: The class `X` is explicitly declared with `__rtti`. This makes it safe to mix the order and type of classes.

In the following example, class `X` inherits only non-polymorphic classes. Class `X` does not need to be declared `__rtti`.

```
struct __rtti S1 { }; // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { }; // The order is not essential
```

Neither the `__rtti` keyword, nor enabling RTTI will make a static class into a polymorphic class.

Runtime Type Identification And Destructors

When destructor cleanup is enabled, a pointer to a class with a virtual destructor can't be deleted if that class is not compiled with runtime type identification enabled. The runtime type identification and destructor cleanup options are on by default. They can be disabled from the C++ page of the Project Options dialog box, or by using the `-xd-` and `-RT-` command-line options.

Example

```
class Alpha {
public:
    virtual ~Alpha( ) { }
};
void func( Alpha *Aptr ) {
    delete Aptr;           // Error.  Alpha is not a polymorphic class type
}
```

Referencing

This section contains Typeid Operator topics.

In This Section

[Referencing](#)

[Simple References](#)

[Reference Arguments](#)

Referencing

In the C programming language, you can pass arguments only by value. In C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects. See the following topics for a discussion of referencing.

Simple references

Reference arguments

Note: C++ specific pointer referencing and dereferencing is discussed in C++ specific operators.

Simple References

The reference declarator can be used to declare references outside functions:

```
int i = 0;
int &ir = i;    // ir is an alias for i
ir = 2;        // same effect as i = 2
```

This creates the lvalue `ir` as an alias for `i`, provided the initializer is the same type as the reference. Any operations on `ir` have precisely the same effect as operations on `i`. For example, `ir = 2` assigns 2 to `i`, and `&ir` returns the address of `i`.

Reference Arguments

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir);    // ir is type "reference to int"
.
.
.
int sum = 3;
func1(sum);              // sum passed by value
func2(sum);              // sum passed by reference
```

The `sum` argument passed by reference can be changed directly by `func2`. On the other hand, `func1` gets a copy of the `sum` argument (passed by value), so `sum` itself cannot be altered by `func1`.

When an actual argument `x` is passed by value, the matching formal argument in the function receives a copy of `x`. Any changes to this copy within the function body are not reflected in the value of `x` outside the scope of the function. Of course, the function can return a value that could be used later to change `x`, but the function cannot directly alter a parameter passed by value.

In C, changing the value of a function parameter outside the scope of the function requires that you pass the address of the parameter. The address is passed by value, thus changing the contents of the address effects the value of the parameter outside the scope of the function.

Even if the function does not need to change the value of a parameter, it is still useful to pass the address (or a reference) to a function. This is especially true if the parameter is a large data structure or object. Passing an object directly to a function necessitates copying the entire object.

Compare the three implementations of the function treble:

Implementation 1

```
int treble_1(int n)
{
    return 3 * n;
}

.
.
.
int x, i = 4;
x = treble_1(i); // x now = 12, i = 4
.
.
.
```

Implementation 2

```
void treble_2(int* np)
{
    *np = (*np) * 3;
}

.
.
.
treble_2(&i); // i now = 12
```

Implementation 3

```
void treble_3(int& n) // n is a reference type
{
    n = n * 3;
}

.
.
.
treble_3(i); // i now = 36
```

The formal argument declaration `type& t` establishes `t` as type “reference to type.” So, when `treble_3` is called with the real argument `i`, `i` is used to initialize the formal reference argument `n`. `n` therefore acts as an alias for `i`, so `n = n*3` also assigns `3 * i` to `i`.

If the initializer is a constant or an object of a different type than the reference type, creates a temporary object for which the reference acts as an alias:


```
int& ir = 6;    /* temporary int object created, aliased by ir, gets value 6 */
float f;
int& ir2 = f;   /* creates temporary int object aliased by ir2; f converted
                before assignment */
ir2 = 2.0       // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

The Scope Resolution Operator

This section contains Scope Resolution Operator topics.

In This Section

[Scope Resolution Operator ::](#)

Scope Resolution Operator ::

The scope access (or resolution) operator :: (two colons) lets you access a global (or file duration) member name even if it is hidden by a local redeclaration of that name. You can use a global identifier by prefixing it with the scope resolution operator. You can access a nested member name by specifying the class name and using the scope resolution operator. Therefore, Alpha::func() and Beta::func() are two different functions.

The new And delete Operators

This section contains new And delete Operator topics.

In This Section

[new](#)

[delete](#)

[Operator new Placement Syntax](#)

[Handling Errors For The New Operator](#)

[The Operator new With Arrays](#)

[The delete Operator With Arrays](#)

[Operator new](#)

[Overloading The Operator new](#)

[Overloading The Operator delete](#)

new

Category

Operators, C++-Specific Keywords

Syntax

```
void *operator new(std::size_t size) throw(std::bad_alloc);
void *operator new(std::size_t size, const std::nothrow_t &) throw();
void *operator new[](std::size_t size) throw(std::bad_alloc);
void *operator new[](std::size_t size, const std::nothrow_t &) throw();
void *operator new(std::size_t size, void *ptr) throw(); // Placement form
void *operator new[](std::size_t size, void *ptr) throw(); // Placement form
```

Description

The new operators offer dynamic storage allocation, similar but superior to the standard library function malloc. These allocation functions attempt to allocate size bytes of storage. If successful, new returns a pointer to the allocated memory. If the allocation fails, the new operator will call the new_handler function. The default behavior of new_handler is to throw an exception of type bad_alloc. If you do not want an exception to be thrown, use the nothrow version of operator new. The nothrow versions return a null pointer result on failure, instead of throwing an exception.

The default placement forms of operator new are reserved and cannot be redefined. You can, however, overload the placement form with a different signature (i.e. one having a different number, or different type of arguments). The default placement forms accept a pointer of type void, and perform no action other than to return that pointer, unchanged. This can be useful when you want to allocate an object at a known address. Using the placement form of new can be tricky, as you must remember to explicitly call the destructor for your object, and then free the pre-allocated memory buffer. Do not call the delete operator on an object allocated with the placement new operator.

A request for non-array allocation uses the appropriate operator new() function. Any request for array allocation will call the appropriate operator new[]() function. Remember to use the array form of operator delete[](), when deallocating an array created with operator new[]().

Note: Arrays of classes require that a default constructor be defined in the class.

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

delete

Category

Operators, C++-Specific Keywords

Syntax

```
void operator delete(void *ptr) throw();  
void operator delete(void *ptr, const std::nothrow_t&) throw();  
void operator delete[](void *ptr) throw();  
void operator delete[](void *ptr, const std::nothrow_t &) throw();  
void operator delete(void *ptr, void *) throw(); // Placement form  
void operator delete[](void *ptr, void *) throw(); // Placement form
```

Description

The delete operator deallocates a memory block allocated by a previous call to new. It is similar but superior to the standard library function free.

You should use the delete operator to remove all memory that has been allocated by the new operator. Failure to free memory can result in memory leaks.

The default placement forms of operator delete are reserved and cannot be redefined. The default placement delete operator performs no action (since no memory was allocated by the default placement new operator). If you overload the placement version of operator new, it is a good idea (though not strictly required) to provide the overload the placement delete operator with the corresponding signature.

Operator new Placement Syntax

The placement syntax for operator new() can be used only if you have overloaded the allocation operator with the appropriate arguments. You can use the placement syntax when you want to use and reuse a memory space which you set up once at the beginning of your program.

When you use the overloaded operator new() to specify where you want an allocation to be placed, you are responsible for deleting the allocation. Because you call your version of the allocation operator, you cannot depend on the global ::operator delete() to do the cleanup.

To release memory, you make an explicit call on the destructor. This method for cleaning up memory should be used only in special situations and with great care. If you make an explicit call of a destructor before an object that has been constructed on the stack goes out of scope, the destructor will be called again when the stackframe is cleaned up.

Handling Errors For The New Operator

By default, new throws the bad_alloc exception when a request for memory allocation cannot be satisfied.

You can define a function to be called if the new operator fails. To tell the new operator about the new-handler function, use set_new_handler and supply a pointer to the new-handler. If you want new to return null on failure, you must use set_new_handler(0) .

The Operator new With Arrays

When using the array form of operator new[](), the pointer returned points to the first element of the array. When creating multidimensional arrays with new, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];    // OK
mat_ptr = new int[n][10][12];    // OK
mat_ptr = new int[3][][12];      // illegal
mat_ptr = new int[][10][12];     // illegal
```

Although the first array dimension can be a variable, all following dimensions must be constants.

The delete Operator With Arrays

Arrays are deleted by operator `delete[]()`. You must use the syntax `delete [] expr` when deleting an array.

```
char * p;
void func()
{
    p = new char[10];    // allocate 10 chars
    delete[] p;         // delete 10 chars
}
```

Early C++ compilers required the array size to be named in the delete expression. In order to handle legacy code, the compiler issues a warning and simply ignores any size that is specified. For example, if the preceding example reads `delete[10] p` and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

Operator new

By default, if there is no overloaded version of `new`, a request for dynamic memory allocation always uses the global version of `new`, `::operator new()`. A request for array allocation calls `::operator new[]()`. With class objects of type name, a specific operator called `name::operator new()` or `name::operator new[]()` can be defined. When `new` is applied to class name objects it invokes the appropriate `name::operator new` if it is present; otherwise, the global `::operator new` is used.

Only the operator `new()` function will accept an optional initializer. The array allocator version, `operator new[]()`, will not accept initializers. In the absence of explicit initializers, the object created by `new` contains unpredictable data (garbage). The objects allocated by `new`, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the default constructor. The user-defined `new` operator with customized initialization plays a key role in C++ constructors for class-type objects.

Overloading The Operator new

The global `::operator new()` and `::operator new[]()` can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program.

Class-specific memory allocation operators can also be overloaded. The operator `new` can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator `new` must return a `void*` and must have a `size_t` as its first argument. To overload the `new` operators, use the following prototypes declared in the `new.h` header file.

- `void * operator new(size_t Type_size);` // For Non-array

- `void * operator new[](size_t Type_size); // For arrays`

The compiler provides `Type_size` to the `new` operator. Any data type may be substituted for `Type_size` except function names (although a pointer to function is permitted), class declarations, enumeration declarations, `const`, `volatile`.

Overloading The Operator `delete`

The global operators, `::operator delete()`, and `::operator delete[]()` cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the each global `delete` function can exist in the program.

The user-defined operator `delete` must have a `void` return type and `void*` as its first argument; a second argument of type `size_t` is optional. A class `T` can define at most one version of each of `T::operator delete[]()` and `T::operator delete()`. To overload the `delete` operators, use the following prototypes.

- `void operator delete(void *Type_ptr, [size_t Type_size]); // For Non-array`
- `void operator delete[](size_t Type_ptr, [size_t Type_size]); // For arrays`

Classes

This section contains Class topics.

In This Section

[C++ Classes](#)

[CLX and VCL Class Declarations](#)

[Class Names](#)

[Class Types](#)

[Class Name Scope](#)

[Class Objects](#)

[Class Member List](#)

[Member Functions](#)

[The Keyword This](#)

[Inline Functions](#)

[Member Scope](#)

[Virtual Base Classes](#)

[Friends Of Classes](#)

C++ Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that inherit the members of one or more base (or parent) classes.

In C++, structures and unions are considered classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

```
class-key <type-info> class-name
```

```
<: base-list> { <member-list> };
```

class-key is one of class, struct, or union.

The optional type-info indicates a request for runtime type information about the class. You can compile with the –RT compiler option, or you can use the `__rtti` keyword.

The optional base-list lists the base class or classes from which the class class-name will derive (or inherit) objects and methods. If any base classes are specified, the class class-name is called a derived class. The base-list has default and optional overriding access specifiers that can modify the access rights of the derived class to members of the base classes.

The optional member-list declares the class members (data and functions) of class-name with default and optional overriding access specifiers that can affect which functions can access which members.

CLX and VCL Class Declarations

Syntax

```
__declspec(<decl-modifier>)
```

Description

The decl-modifier argument can be `delphiclass` or `pascalimplementation`. These arguments should be used only with classes derived from VCL classes.

- You must use `__declspec(delphiclass)` for any forward declaration of classes that are directly or indirectly derived from `TObject`.
- Use the `__declspec(pascalimplementation)` modifier to indicate that a class has been implemented in the Delphi language. This modifier appears in a Delphi portability header file with a `.hpp` extension.

Note: Another argument, `delphireturn`, is used internally to mark C++ classes for VCL-compatible handling in function calls as parameters and return values.

The `delphiclass` argument is used to create classes that have the following VCL compatibility.

- VCL-compatible RTTI
- VCL-compatible constructor/destructor behavior
- VCL-compatible exception handling

A VCL-compatible class has the following restrictions.

- No virtual base classes or multiple inheritance is allowed.
- Must be dynamically allocated by using the global `new` operator.
- Copy and assignment constructors must be explicitly defined. The compiler does not automatically provide these constructors for VCL-derived classes.
- Must publicly inherit from another VCL class.

Class Names

class-name is any identifier unique within its scope. With structures, classes, and unions, class-name can be omitted. See Untagged structures and typedefs for discussion of untagged structures.

Class Types

The declaration creates a unique type, class type class-name. This lets you declare further class objects (or instances) of this type, and objects derived from this type (such as pointers to, references to, arrays of class-name, and so on):

```
class X { ... };
X x, &xr, *xpтр, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of X */
struct Y { ... };
Y y, &yr, *ypтр, yarray[10];
// C would have
// struct Y y, *ypтр, yarray[10];
union Z { ... };
Z z, &zr, *zpтр, zarray[10];
// C would have
// union Z z, *zpтр, zarray[10];
```

Note the difference between C and C++ structure and union declarations: The keywords `struct` and `union` are essential in C, but in C++, they are needed only when the class names, `Y` and `Z`, are hidden (see Class name scope)

Class Name Scope

The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class

name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the elaborated type specifier. This means that the class key, class, struct, or union, must be used with the class name. For example,

```
struct S { ... };
int S(struct S *Sptr);
void func(void) {
    S t;           // ILLEGAL declaration: no class key and function S in scope
    struct S s;    // OK: elaborated with class key
    S(&s);         // OK: this is a function call
}
```

C++ also allows a forward class declaration:

```
class X; // no members, yet!
```

Forward declarations permit certain references to class name X (usually references to pointers to class objects) before the class has been fully defined. See Structure member declarations for more information. Of course, you must make a complete class declaration with members before you can define and use objects of that class.

See also the syntax for forward declarations of CLXVCL classes.

Class Objects

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class.

Redefined functions and operators are said to be overloaded. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called member functions for that class. C++ offers the overloading mechanism that allows the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

Class Member List

The optional member-list is a sequence including, but not exclusive to:

- Data declarations (of any type, including enumerations, bit fields and other classes)
- Nested type declarations
- Nested type definitions
- Template declarations
- Template definitions
- Function declarations
- Function definitions
- Constructors
- A destructor

Members of a class can optional have storage class specifiers and access modifiers. The objects thus defined are called class members. The storage class specifiers auto, extern, and register are not allowed. Members can be declared with the static storage class specifiers.

Member Functions

A function declared without the friend specifier is known as a member function of the class. Functions declared with the friend modifier are called friend functions.

Member functions are often referred to as methods in Delphi documentation.

The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.

The Keyword This

This section contains Keyword this topics.

In This Section

[Static Members](#)

Static Members

The storage class specifier `static` can be used in class declarations of data and function members. Such members are called static members and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If `x` is a static member of class `X`, it can be referenced as `X::x` (even if objects of class `X` haven’t been created yet). It is still possible to access `x` using the normal member access operators. For example, `y.x` and `yptr->x`, where `y` is an object of class `X` and `yptr` is a pointer to an object of class `X`, although the expressions `y` and `yptr` are not evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
    int member_int;
public:
    static void func(int i, X* ptr);
};

void g(void)
{
    X obj;
    func(1, &obj);           // error unless there is a global func()
                             // defined elsewhere
    X::func(1, &obj);         // calls the static func() in X
                             // OK for static functions only
    obj.func(1, &obj);        // so does this (OK for static and
                             // nonstatic functions)
}
```

Because static member functions can be called with no particular object in mind, they don’t have a `this` pointer, and therefore cannot access nonstatic members without explicitly specifying an object with `.` or `->`. For example, with the declarations of the previous example, `func` might be defined as follows:

```
void X::func(int i, X* ptr)
{
    member_int = i;          // which object does member_int
                             // refer to? Error
    ptr->member_int = i;     // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members, nested to any level, obey the usual class member access rules, except they can be initialized.

```

class X {
    static int x;
    static const int size = 5;
    class inner {
        static float f;
        void func(void);    // nested declaration
    };
public :
    char array[size];
};
int X::x = 1;
float X::inner::f = 3.14; // initialization of nested static
void X::inner::func(void) {    /* define the nested function */ }

```

The principal use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- Reduce the number of visible global names
- Make obvious which static objects logically belong to which class
- Permit access control to their names

Inline Functions

This section contains Inline Function topics.

In This Section

[Inline Functions](#)

Inline Functions

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an inline function.

The compiler can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an inline expansion of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The inline specifier indicates to the compiler you would like an inline expansion.

Note: The compiler can ignore requests for inline expansion.

Explicit and implicit inline requests are best reserved for small, frequently used functions, such as the operator functions that implement overloaded operators. For example, the following class declaration of func:

```
int i;                                // global int
class X {
public:
    char* func(void) { return i; }    // inline by default
    char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined outside the class with an explicit inline specifier. The value i returned by func is the char* i of class X (see Member scope).

Inline functions and exceptions

An inline function with an exception-specification will never be expanded inline by the compiler. For example,

```
inline void f1() throw(int)
{
    // Warning: Functions with exception specifications are not expanded inline
}
```

The remaining restrictions apply only when destructor cleanup is enabled.

Note: Destructors are called by default. See Setting exception handling options for information about exception-handling switches.

An inline function that takes at least one parameter that is of type 'class with a destructor' will not be expanded inline. Note that this restriction does not apply to classes that are passed by reference. Example:

```
struct foo {
    foo();
    ~foo();
};
```

```

inline void f2(foo& x)
{
    // no warning, f2() can be expanded inline
}
inline void f3(foo x)
{
    // Warning: Functions taking class-by-value argument(s) are
    //           not expanded inline in function f3(foo)
}

```

An inline function that returns a class with a destructor by value will not be expanded inline whenever there are variables or temporaries that need to be destructed within the return expression:

```

struct foo {
    foo();
    ~foo();
};
inline foo f4()
{
    return foo();
    // no warning, f4() can be expanded inline
}
inline foo f5()
{
    foo X;
    return foo(); // Object X needs to be destructed
    // Warning: Functions containing some return statements are
    //           not expanded inline in function f5()
}
inline foo f6()
{
    return ( foo(), foo() ); // temporary in return value
    // Warning: Functions containing some return statements are
    //           not expanded inline in function f6()
}

```


Member Scope

This section contains Member Scope topics.

In This Section

[Member Scope](#)

[Nested Types](#)

[Member Access Control](#)

[Base And Derived Class Access](#)

Member Scope

The expression `X::func()` in the example in [Inline functions and exceptions](#) uses the class name `X` with the scope access modifier to signify that `func`, although defined “outside” the class, is indeed a member function of `X` and exists within the scope of `X`. The influence of `X::` extends into the body of the definition. This explains why the `i` returned by `func` refers to `X::i`, the `char* i` of `X`, rather than the global `int i`. Without the `X::` modifier, the function `func` would represent an ordinary non-class function, returning the global `int i`.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class `X` can be referenced using the selection operators `.` and `->` (as with C structures). Member functions can also be called using the selection operators (see [The keyword this](#)). For example:

```
class X {
public:
    int i;
    char name[20];
    X *ptr1;
    X *ptr2;
    void Xfunc(char*data, X* left, X* right);    // define elsewhere
};
void f(void);
{
    X x1, x2, *xptr=&x1;
    x1.i = 0;
    x2.i = x1.i;
    xptr->i = 1;
    x1.Xfunc("stan", &x2, xptr);
}
```

If `m` is a member or base member of class `X`, the expression `X::m` is called a qualified name; it has the same type as `m`, and it is an lvalue only if `m` is an lvalue. It is important to note that, even if the class name `X` is hidden by a non-type name, the qualified name `X::m` will access the correct class member, `m`.

Class members cannot be added to a class by another section of your program. The class `X` cannot contain objects of class `X`, but can contain pointers or references to objects of class `X` (note the similarity with C’s structure and union types).

Nested Types

Tag or typedef names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the `xxx::yyy` notation, except when in the scope of the appropriate class.

A class declared within another class is called a nested class. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).

Classes can be nested in this way to an arbitrary level, up to the limits of memory. Nested classes can be declared inside some class and defined later. For example,

```
struct outer
{
    typedef int t;    // 'outer::t' is a typedef name
    struct inner      // 'outer::inner' is a class
    {
        static int x;
    };
    static int x;
    int f();
    class deep;       // nested declaration
};
int outer::x;         // define static data member
int outer::f() {
    t x;              // 't' visible directly here
    return x;
}
int outer::inner::x;  // define static data member
outer::t x;           // have to use 'outer::t' here
class outer::deep { }; // define the nested class here
```

With Borland C++ 2.0, any tags or typedef names declared inside a class actually belong to the global (file) scope. For example:

```
struct foo
{
    enum bar { x };    // 2.0 rules: 'bar' belongs to file scope
                      // 2.1 rules: 'bar' belongs to 'foo' scope
};
bar x;
```

The preceding fragment compiles without errors. But because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Member Access Control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: public, private, and protected. The significance of these attributes is as follows:

- public: The member can be used by any function.
- private: The member can be used only by member functions and friends of the class it's declared in.
- protected: Same as for private. Additionally, the member can be used by member functions and friends of classes derived from the declared class, but only in objects of the derived type. (Derived classes are explained in Base and derived class access.)

Note: Friend function declarations are not affected by access specifiers (see Friends of classes for more information).

Members of a class are private by default, so you need explicit public or protected access specifiers to override the default.

Members of a struct are public by default, but you can override this with the private or protected access specifier.

Members of a union are public by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
    int i;    // X::i is private by default
    char ch;  // so is X::ch
public:
    int j;    // next two are public
    int k;
protected:
    int l;    // X::l is protected
};
struct Y {
    int i;    // Y::i is public by default
private:
    int j;    // Y::j is private
public:
    int k;    // Y::k is public
};
union Z {
    int i;    // public by default; no other choice
    double d;
};
```

Note: The access specifiers can be listed and grouped in any convenient sequence. You can save typing effort by declaring all the private members together, and so on.

Base And Derived Class Access

When you declare a derived class D, you list the base classes B1, B2, ... in a comma-delimited base-list:

```
class-key D : base-list { <member-list> }
```

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) D can use only the public and protected members of its base classes. But, what will be the access attributes of the inherited members as viewed by D? D might want to use a public member from a base class, but make it private as far as outside functions are concerned. The solution is to use access specifiers in the base-list.

Note: Since a base class can itself be a derived class, the access attribute question is recursive: you backtrack until you reach the basemost of the base classes, those that do not inherit.

When declaring D, you can use the access specifier public, protected, or private in front of the classes in the base-list:

```
class D : public B1, private B2, ... {
    .
    .
    .
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they can alter the access attributes of base members as viewed by the derived class.

The default is private if D is a class declaration, and public if D is a struct declaration.

Note: Unions cannot have base classes, and unions cannot be used as base classes.

The derived class inherits access attributes from a base class as follows:

- public base class: public members of the base class are public members of the derived class. protected members of the base class are protected members of the derived class. private members of the base class remain private to the base class.
- protected base class: Both public and protected members of the base class are protected members of the derived class. private members of the base class remain private to the base class.
- private base class: Both public and protected members of the base class are private members of the derived class. private members of the base class remain private to the base class.

Note that private members of a base class are always inaccessible to member functions of the derived class unless friend declarations are explicitly declared in the base class granting access. For example,

```
/* class X is derived from class A */
class X : A {           // default for class is private A
    .
    .
    .
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */
class Y : B, public C { // override default for C
    .
    .
    .
}
/* struct S is derived from D */
struct S : D {          // default for struct is public D
    .
    .
    .
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
struct T : private D, E { // override default for D
                        // E is public by default
    .
    .
    .
}
```

The effect of access specifiers in the base list can be adjusted by using a qualified-name in the public or protected declarations of the derived class. For example:

```
class B {
    int a;           // private by default
public:
    int b, c;
    int Bfunc(void);
};
class X : private B { // a, b, c, Bfunc are now private in X
    int d;           // private by default, NOTE: a is not
```

```

// accessible in X
public:
    B::c;           // c was private, now is public
    int e;
    int Xfunc(void);
};
int Efunc(X& x);    // external to B and X

```

The function Efunc() can use only the public names c, e, and Xfunc().

The function Xfunc() is in X, which is derived from private B, so it has access to

- The “adjusted-to-public” c
- The “private-to-X” members from B: b and Bfunc()
- X’s own private and public members: d, e, and Xfunc()

However, Xfunc() cannot access the “private-to-B” member, a.

Virtual Base Classes

This section contains Virtual Base Class topics.

In This Section

[Virtual Base Classes](#)

Virtual Base Classes

A virtual class is a base class that is passed to more than one derived class, as might happen with multiple inheritance.

You cannot specify a base class more than once in a derived class:

```
class B { ...};  
class D : B, B { ... }; // ILLEGAL
```

However, you can indirectly pass a base class to the derived class more than once:

```
class X : public B { ... };
```

```
class Y : public B { ... };
```

```
class Z : public X, public Y { ... }; // OK
```

In this case, each object of class Z has two sub-objects of class B.

If this causes problems, add the keyword `virtual` to the base class specifier. For example,

```
class X : virtual public B { ... };
```

```
class Y : virtual public B { ... };
```

```
class Z : public X, public Y { ... };
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes.

If the hierarchy contains multiple virtual base classes, the virtual base class constructors invoke in the order they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly constructed. For example, this code

```
class X : public Y, virtual public Z
```

```
X one;
```

produces this order:

```
Z(); // virtual base class initialization
```

```
Y(); // non-virtual base class
```

```
X(); // derived class
```


Friends Of Classes

This section contains Friends Of Class topics.

In This Section

[Friends Of Classes](#)

Friends Of Classes

A friend F of a class X is a function or class, although not a member function of X, with full access rights to the private and protected members of X. In all other respects, F is a normal function with respect to scope, declarations, and definitions.

Since F is not a member of X, it is not in the scope of X, and it cannot be called with the x.F and xptr->F selector operators (where x is an X object and xptr is a pointer to an X object).

If the specifier friend is used with a function declaration or definition within the class X, it becomes a friend of X.

friend functions defined within a class obey the same inline rules as member functions (see Inline functions). friend functions are not affected by their position within the class or by any access specifiers. For example:

```
class X {
    int i;                      // private to X
    friend void friend_func(X*, int);
    /* friend_func is not private, even though it's declared in the private section */
public:
    void member_func(int);
};
/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }
X xobj;
/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class Y into friends of class X with a single declaration:

```
class Y;                        // incomplete declaration
class X {
    friend Y;
    int i;
    void member_funcX();
};
class Y; {                      // complete the declaration
    void friend_X1(X&);
    void friend_X2(X*);
    .
    .
    .
};
```

The functions declared in Y are friends of X, although they have no friend specifiers. They can access the private members of X, such as i and member_funcX.

It is also possible for an individual member function of class X to be a friend of class Y:

```
class X {  
.  
.  
.  
    void member_funcX();  
}  
class Y {  
    int i;  
    friend void X::member_funcX();  
.  
.  
.  
};
```

Class friendship is not transitive: X friend of Y and Y friend of Z does not imply X friend of Z. Friendship is not inherited.

Constructors And Destructors

This section contains Constructor and Destructor topics.

In This Section

[Introduction To Constructors And Destructors](#)

[Constructors](#)

[Destructors](#)

Introduction To Constructors And Destructors

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features:

- They do not have return value declarations (not even void).
- They cannot be inherited, though a derived class can call the base class's constructors and destructors.
- Constructors, like most C++ functions, can have default arguments or use member initialization lists.
- Destructors can be virtual, but constructors cannot. (See Virtual destructors.)
- You can't take their addresses.

```
int main (void)
{
    .
    .
    .
    void *ptr = base::base;    // illegal
    .
    .
    .
}
```

- Constructors and destructors can be generated by the compiler if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.
- You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```

{
.
.
.
X *p;
.
.
.
p->X::~~X();           // legal call of destructor
X::X();                // illegal call of constructor
.
.
.
}

```

- The compiler automatically calls constructors and destructors when defining and destroying objects.
- Constructors and destructors can make implicit calls to operator new and operator delete if allocation is required for an object.
- An object with a constructor or destructor cannot be used as a member of a union.
- If no constructor has been defined for some class X to accept a given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class X. Note that this rule applies only to any constructor with one parameter and no initializers that use the “=” syntax.

```

class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;           // illegal: Y(X(1)) not tried

```

If class X has one or more constructors, one of them is invoked each time you define an object x of class X. The constructor creates x and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

Constructors

This section contains Constructor topics.

In This Section

[Constructors](#)

[Constructor Defaults](#)

[The Copy Constructor](#)

[Overloading Constructors](#)

[Order Of Calling Constructors](#)

[Class Initialization](#)

Constructors

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the main function is called. Global variable constructors are also called prior to #pragma startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {
public:
    X();    // class X constructor
};
```

A class X constructor cannot take X as an argument:

```
class X {
public:
    X(X);    // illegal
};
```

The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the copy constructor. A constructor that accepts no parameters is called the default constructor.

Constructor Defaults

The default constructor for class X is one that takes no arguments; it usually has the form X::X(). If no user-defined constructors exist for a class, the compiler generates a default constructor. On a declaration such as X x, the default constructor creates the object x.

Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero int. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes no arguments and must not be confused with, say, `X::X(int = 0)`, which can be called with no arguments as a default constructor, or can take an argument.

You should avoid ambiguity in defining constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
    X();
    X(int i = 0);
};
int main()
{
    X one(10); // OK; uses X::X(int)
    X two;     // Error; ambiguous whether to call X::X() or
               // X::X(int = 0)
    return 0;
}
```

The Copy Constructor

A copy constructor for class `X` is one that can be called with a single argument of type `X` as follows:

```
X::X(X&)
```

or

```
X::X(const X&)
```

or

```
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x1;
X x2 = x1;
X x3(x1);
```

The compiler generates a copy constructor for class `X` if one is needed and no other constructor has been defined in class `X`. The copy constructor that is generated by the compiler lets you safely start programming with simple data types. You need to make your own definition of the copy constructor if your program creates aggregate, complex types such as class, struct, and array types. The copy constructor is also called when you pass a class argument by value to a function.

See also the discussion of member-by-member class assignment. You should define the copy constructor if you overload the assignment operator.

Overloading Constructors

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```

class X {
    int    integer_part;
    double double_part;
public:
    X(int i)    { integer_part = i; }
    X(double d) { double_part = d; }
};
int main()
{
    X one(10);    // invokes X::X(int) and sets integer_part to 10
    X one(3.14); // invokes X::X(double) setting double_part to 3.14
    return 0;
}

```

Order Of Calling Constructors

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```

class Y {...}
class X : public Y {...}
X one;

```

the constructors are called in this order:

```

Y();    // base class constructor
X();    // derived class constructor

```

For the case of multiple base classes,

```

class X : public Y, public Z
X one;

```

the constructors are called in the order of declaration:

```

Y();    // base class constructors come first
Z();
X();

```

Constructors for virtual base classes are invoked before any nonvirtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any nonvirtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a nonvirtual base, that nonvirtual base will be first so that the virtual base class can be properly constructed. The code:

```

class X : public Y, virtual public Z
X one;

```

produces this order:

```
Z();    // virtual base class initialization
Y();    // nonvirtual base class
X();    // derived class
```

Or, for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base();      // virtual base class highest in hierarchy
             // base is constructed only once
base2();     // nonvirtual base of virtual base level2
             // must be called to construct level2
level2();    // virtual base class
base2();     // nonvirtual base of level1
level1();    // other nonvirtual base
toplevel();
```

If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

Class Initialization

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
    int i;
public:
    X();           // function bodies omitted for clarity
    X(int x);
    X(const X&);
};
void main()
{
    X one;         // default constructor invoked
    X two(1);      // constructor X::X(int) is used
    X three = 1;   // calls X::X(int)
    X four = one;  // invokes X::X(const X&) for copy
```



```

    X five(two); // calls X::X(const X&)
}

```

The constructor can assign values to its members in two ways:

1. It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```

class X
{
    int a, b;
public:
    X(int i, int j) { a = i; b = j }
};

```

2. An initializer list can be used prior to the function body:

```

class X
{
    int a, b, &c; // Note the reference variable.
public:
    X(int i, int j) : a(i), b(j), c(a) {}
};

```

The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to `x::a` and 2 to `x::b`. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

Note: Base class constructors must be declared as either public or protected to be called from a derived class.

```

class base1
{
    int x;
public:
    base1(int i) { x = i; }
};
class base2
{
    int x;
public:
    base2(int i) : x(i) {}
};
class top : public base1, public base2
{
    int a, b;
public:
    top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j;}
};

```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of `base1` with the value 5 and `base2` with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```

class X
{
    int a, b;

```

```
public:
    X(int i, j) : a(i), b(a+j) {}
};
```

With this class, a declaration of `X x(1,1)` results in an assignment of 1 to `x::a` and 2 to `x::b`.

Base class constructors are called prior to the construction of any of the derived classes members. If the values of the derived class are changed, they will have no effect on the creation of the base class.

```
class base
{
    int x;
public:
    base(int i) : x(i) {}
};
class derived : base
{
    int a;
public:
    derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                         // passed an uninitialized 'a'
};
```

With this class setup, a call of `derived d(1)` will not result in a value of 10 for the base class member `x`. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
    .
    .
    .
}
```

Destructors

This section contains Destructor topics.

In This Section

[Destructors](#)

[Invoking Destructors](#)

[Atexit, #pragma Exit, And Destructors](#)

[Exit And Destructors](#)

[Abort And Destructors](#)

[Virtual Destructors](#)

Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
    ~X() {}; // destructor for class X
};
```

If a destructor isn't explicitly defined for a class, the compiler generates one.

Invoking Destructors

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the delete operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see Order of calling constructors).

Atexit, #pragma Exit, And Destructors

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the main function, are destroyed as they go out of scope. The order of execution at the end of a program is as follows:

- atexit() functions are executed in the order they were inserted.
- #pragma exit functions are executed in the order of their priority codes.
- Destructors for global variables are called.

Exit And Destructors

When you call exit from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

Abort And Destructors

If you call abort anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to delete, or directly by using the destructor's fully qualified name. You can use delete to destroy objects that have been allocated using new. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to new

```
#include <stdlib.h>
class X {
public:
    .
    .
    .
    ~X() {};
    .
    .
    .
};
void* operator new(size_t size, void *ptr)
{
    return ptr;
}
char buffer[sizeof(X)];
void main()
{
    X* pointer = new X;
    X* exact_pointer;
    exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer
    .
    .
    .
    delete pointer;                // delete used to destroy pointer
    exact_pointer->X::~~X();         // direct call used to deallocate
}
```

Virtual Destructors

A destructor can be declared as virtual. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a virtual destructor is itself virtual.

```
/* How virtual affects the order of destructor calls.
   Without a virtual destructor in the base class, the derived
   class destructor won't be called. */
#include <iostream>
class color {
public:
    virtual ~color() { // Virtual destructor
        std::cout << "color dtor\n";
    }
}
```

```

    }
};
class red : public color {
public:
    ~red() { // This destructor is also virtual
        std::cout << "red dtor\n";
    }
};
class brightred : public red {
public:
    ~brightred() { // This destructor is also virtual
        std::cout << "brightred dtor\n";
    }
};
int main()
{
    color *palette[3];
    palette[0] = new red;
    palette[1] = new brightred;
    palette[2] = new color;
    // The destructors for red and color are called.
    delete palette[0];
    std::cout << std::endl;
    // The destructors for bright red, red, and color are called.
    delete palette[1];
    std::cout << std::endl;
    // The destructor for color is called.
    delete palette[2];
    return 0;
}

```

Program Output:

```

red dtor
color dtor
brightred dtor
red dtor
color dtor
color dtor

```

However, if no destructors are declared as virtual, `delete palette[0]`, `delete palette[1]`, and `delete palette[2]` would all call only the destructor for class `color`. This would incorrectly destruct the first two elements, which were actually of type `red` and `brightred`.

Operator Overloading Overview

This section contains Operator Overloading Overview topics.

In This Section

[Overloading Operators](#)

[How To Construct A Class Of Complex Vectors](#)

Overloading Operators

C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators can be overloaded except for:

```
.  .*  ::  ?:
```

The following preprocessing symbols cannot be overloaded.

```
#  ##
```

The =, [], (), and -> operators can be overloaded only as nonstatic member functions. These operators cannot be overloaded for enum types. Any attempt to overload a global version of these operators results in a compile-time error.

The keyword operator followed by the operator symbol is called the operator function name; it is used like a normal function name when defining the new (overloaded) action for the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function cannot alter the number of arguments or the precedence and associativity rules applying to normal operator use.

How To Construct A Class Of Complex Vectors

This section contains How to Construct A Class Of Complex Vector topics.

In This Section

[Example Of Overloading Operators](#)

Example Of Overloading Operators

The following example extends the class `complex` to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in the usual mathematical syntax.

Some of the issues illustrated by the example are:

- The default constructor is defined. The default constructor is provided by the compiler only if you have not defined it or any other constructor.
- The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator.
- The assignment operator is overloaded. If you do not overload the assignment operator, the compiler calls a default assignment operator when required. By overloading assignment of `cvector` types, you specify exactly the actions to be taken. Note that derived classes cannot inherit the assignment operator.
- The subscript operator is defined as a member function (a requirement when overloading) with a single argument. The `const` version assures the caller that it will not modify its argument—this is useful when copying or assigning. This operator should check that the index value is within range—a good place to implement exception handling.
- The addition operator is defined as a member function. It allows addition only for `cvector` types. Addition should always check that the operands' sizes are compatible.
- The multiplication operator is declared a friend. This lets you define the order of the operands. An attempt to reverse the order of the operands is a compile-time error.
- The stream insertion operator is overloaded to naturally display a `cvector`. Large objects that don't display well on a limited size screen might require a different display strategy.

Example Source

```
/* HOW TO EXTEND THE complex CLASS AND OVERLOAD THE REQUIRED OPERATORS. */
complexcomplexcomplex
#include <complex> // This includes iostream
using namespace std;
// COMPLEX VECTORS
template <class T>
class cvector {
    int size;
    complex<T> *data;
public:
    cvector() { size = 0; data = NULL; };
    cvector(int i = 5) : size(i) { // DEFAULT VECTOR SIZE.
        data = new complex<T>[size];
        for (int j = 0; j < size; j++)
            data[j] = j + (0.1 * j); // ARBITRARY INITIALIZATION.
    };
    /* THIS VERSION IS CALLED IN main() */
    complex<T>& operator [](int i) { return data[i]; };
    /* THIS VERSION IS CALLED IN ASSIGNMENT OPERATOR AND COPY THE CONSTRUCTOR */
    const complex<T>& operator [](int i) const { return data[i]; };
    cvector operator +(cvector& A) { // ADDITION OPERATOR
```

```

    cvector result(A.size); // DO NOT MODIFY THE ORIGINAL
    for (int i = 0; i < size; i++)
        result[i] = data[i] + A.data[i];
    return result;
};

/* BECAUSE scalar * vector MULTIPLICATION IS NOT COMMUTATIVE, THE ORDER OF
THE ELEMENTS MUST BE SPECIFIED. THIS FRIEND OPERATOR FUNCTION WILL ENSURE
PROPER MULTIPLICATION. */
friend cvector operator *(T scalar, cvector& A) {
    cvector result(A.size); // DO NOT MODIFY THE ORIGINAL
    for (int i = 0; i < A.size; i++)
        result.data[i] = scalar * A.data[i];
    return result;
}

/* THE STREAM INSERTION OPERATOR. */
friend ostream& operator <<(ostream& out_data, cvector& C) {
    for (int i = 0; i < C.size; i++)
        out_data << "[" << i << "]= " << C.data[i] << "    ";
    cout << endl;
    return out_data;
};

cvector( const cvector &C ) { // COPY CONSTRUCTOR
    size = C.size;
    data = new complex<T>[size];
    for (int i = 0; i < size; i++)
        data[i] = C[i];
}

cvector& operator =(const cvector &C) { // ASSIGNMENT OPERATOR.
    if (this == &C) return *this;
    delete[] data;
    size = C.size;
    data = new complex<T>[size];
    for (int i = 0; i < size; i++)
        data[i] = C[i];
    return *this;
};

virtual ~cvector() { delete[] data; }; // DESTRUCTOR
};

int main(void) { /* A FEW OPERATIONS WITH complex VECTORS. */
    cvector<float> cvector1(4), cvector2(4), result(4);
    // CREATE complex NUMBERS AND ASSIGN THEM TO complex VECTORS
    cvector1[3] = complex<float>(3.3, 102.8);
    cout << "Here is cvector1:" << endl;
    cout << cvector1;
    cvector2[3] = complex<float>(33.3, 81);
    cout << "Here is cvector2:" << endl;
    cout << cvector2;
    result = cvector1 + cvector2;
    cout << "The result of vector addition:" << endl;
    cout << result;
    result = 10 * cvector2;
    cout << "The result of 10 * cvector2:" << endl;
    cout << result;
    return 0;
}

```

Output

```

Here is cvector1:
[0]=(0, 0)    [1]=(1.1, 0)    [2]=(2.2, 0)    [3]=(3.3, 102.8)

```

```
Here is cvector2:  
[0]=(0, 0)    [1]=(1.1, 0)    [2]=(2.2, 0)    [3]=(33.3, 81)  
The result of vector addition:  
[0]=(0, 0)    [1]=(2.2, 0)    [2]=(4.4, 0)    [3]=(36.6, 183.8)  
The result of 10 * cvector2:  
[0]=(0, 0)    [1]=(11, 0)    [2]=(22, 0)    [3]=(333, 810)
```


Overloading Operator Functions Overview

This section contains Overloading Operator Functions Overview topics.

In This Section

- [Overloading Operator Functions](#)
- [Overloaded Operators And Inheritance](#)
- [Overloading Unary Operators](#)
- [Overloading Binary Operators](#)
- [Overloading The Assignment Operator =](#)
- [Overloading The Function Call Operator \(\)](#)
- [Overloading The Subscript Operator \[\]](#)
- [Overloading The Class Member Access Operators ->](#)

Overloading Operator Functions

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2);    // same as c3 = c1 + c2
```

Apart from new and delete, which have their own rules, an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions =, (), [] and -> must be nonstatic member functions.

Enumerations can have overloaded operators. However, the operator functions =, (), [], and -> cannot be overloaded for enumerations.

Overloaded Operators And Inheritance

With the exception of the assignment function operator =(), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could possibly be further overloaded for Y.

Overloading Unary Operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either x.operator@() or operator@(x), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

- Under C++ 2.0, an overloaded operator ++ or -- is used for both prefix and postfix uses of the operator.
- With C++ 2.1, when an operator ++ or operator -- is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix operator ++ or operator --. You can only overload a postfix operator ++ or operator -- by defining it as a member function taking an int parameter or as a nonmember function taking one class and one int parameter.

When only the prefix version of an operator ++ or operator -- is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator in function func()
Warning: Overloaded prefix 'operator --' used as a postfix operator in function func()
```

Overloading Binary Operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually friend) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either x.operator@(y) or operator@(x,y) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Overloading The Assignment Operator =

The assignment operator=() can be overloaded by declaring a nonstatic member function. For example,

```
class String {
    .
    .
    .
    String& operator = (String& str);
    .
    .
    .
    String (String&);
    ~String();
}
```

This code, with suitable definitions of String::operator =(), allows string assignments str1 = str2 in the usual sense. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class X, there is no user-defined operator =, the operator = is defined by default as a member-by-member assignment of the members of class X:

```
X& X::operator = (const X& source)
{
    // memberwise assignment
}
```

Overloading The Function Call Operator ()

Syntax

```
postfix-expression ( <expression-list> )
```

Description

In its ordinary use as a function call, the postfix-expression must be a function name, or a pointer or reference to a function. When the postfix-expression is used to make a member function call, postfix-expression must be a class member function name or a pointer-to-member expression used to select a class member function. In either case, the postfix-expression is followed by the optional expression-list.

A call X(arg1, arg2), where X is an object class X, is interpreted as X.operator()(arg1, arg2).

The function call operator, operator()(), can only be overloaded as a nonstatic member function.

Overloading The Subscript Operator []

Syntax

```
postfix-expression [ expression ]
```

Description

The corresponding operator function is operator[](()) this can be user-defined for a class X (and any derived classes). The expression X[y], where X is an object of class X, is interpreted as x.operator[](y).

The operator[](()) can only be overloaded as a nonstatic member function.

Overloading The Class Member Access Operators ->

Syntax

```
postfix-expression -> primary-expression
```

Description

The expression x->m, where x is a class X object, is interpreted as (x.operator->())->m, so that the function operator->() must either return a pointer to a class object or return an object of a class for which operator-> is defined.

The operator->() can only be overloaded as a nonstatic member function.

Polymorphic Classes

This section contains Polymorphic Class topics.

In This Section

[Polymorphic Classes](#)

[Virtual Functions](#)

[Dynamic Functions](#)

[Abstract Classes](#)

Polymorphic Classes

Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are class and struct.

Virtual Functions

This section contains Virtual Function topics.

In This Section

[Virtual Functions](#)

Virtual Functions

virtual functions allow derived classes to provide different versions of a base class function. You can use the virtual keyword to declare a virtual function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the virtual keyword. To declare a pure function (which automatically declares an abstract class), prefix the prototype with the virtual keyword, and set the function equal to zero.

```
virtual int funct1(void);           // A virtual function declaration.
virtual int funct2(void) = 0;       // A pure function declaration.
```

A function declaration cannot provide both a pure-specifier and a definition.

Example:

```
struct C {
    virtual void f() { } = 0; // ill-formed
};
```

The only legal syntax to provide a body is:

```
struct TheClass
{
    virtual void funct3(void) = 0;
};
virtual void TheClass::funct3(void)
{
    // Some code here.
};
```

Note: See [Abstract classes](#) for a discussion of pure virtual functions.

When you declare virtual functions, keep these guidelines in mind:

- They can be member functions only.
- They can be declared a friend of another class.
- They cannot be a static member.

A virtual function does not need to be redefined in a derived class. You can supply one definition in the base class so that all calls will access the base function.

To redefine a virtual function in any derived class, the number and type of arguments must be the same in the base class declaration and in the derived class declaration. (The case for redefined virtual functions differing only in return type is discussed below.) A redefined function is said to override the base class function.

You can also declare the functions `int Base::Fun(int)` and `int Derived::Fun(int)` even when they are not virtual. In such a case, `int Derived::Fun(int)` is said to hide any other versions of `Fun(int)` that exist in any base classes. In addition, if class `Derived` defines other versions of `Fun()`, (that is, versions of `Fun()` with different signatures) such versions are said to be overloaded versions of `Fun()`.

Virtual function return types

Generally, when redefining a virtual function, you cannot change just the function return type. To redefine a virtual function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the virtual function mechanism is ignored.

However, for certain virtual functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when both of the following conditions are met:

- The overridden virtual function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class B and class D (derived publicly from B) each contain a virtual function vf, then if vf is called for an object d of D, the call made is D::vf(), even when the access is via a pointer or reference to B. For example,

```
struct X {};// Base class.
struct Y : X {};// Derived class.
struct B {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    void f();
    virtual X* pf();// Return type is a pointer to base. This can
    // be overridden.
};
class D : public B {
public:
    virtual void vf1();// Virtual specifier is legal but redundant.
    void vf2(int);// Not virtual, since it's using a different
    // arg list. This hides B::vf2().
    // char vf3();// Illegal: return-type-only change!
    void f();
    Y* pf();// Overriding function differs only
    // in return type. Returns a pointer to
    // the derived class.
};
void extf()
{
    D d;// Instantiate D
    B* bp = &d;// Standard conversion from D* to B*
    // Initialize bp with the table of functions
    // provided for object d. If there is no entry for a
    // function in the d-table, use the function
    // in the B-table.
    bp->vf1(); // Calls D::vf1
    bp->vf2(); // Calls B::vf2 since D's vf2 has different args
    bp->f(); // Calls B::f (not virtual)
    X* xptr = bp->pf();// Calls D::pf() and converts the result
    // to a pointer to X.
    D* dptr = &d;
    Y* yptr = dptr->pf();// Calls D::pf() and initializes yptr.
    // No further conversion is done.
}
```

The overriding function vf1 in D is automatically virtual. The virtual specifier can be used with an overriding function declaration in the derived class. If other classes will be derived from D, the virtual keyword is required. If no further classes will be derived from D, the use of virtual is redundant.

The interpretation of a virtual function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at runtime (late binding).

Dynamic Functions

dynamic functions are allowed for classes derived from TObject. Dynamic functions are similar to virtual functions except for the way they are stored in the virtual tables. Virtual functions occupy a slot in the virtual table in the object they are defined in, and in the virtual table of every descendant of that object. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of its ancestors are searched until the function is found.

Therefore, dynamic functions reduces the size of your virtual tables at the expense of a delay at runtime to look up the address of the functions.

Because dynamic functions are available only in classes derived from TObject, you will get an error if you use them in a regular class. For example:

```
class dynfunc {
int __declspec(dynamic) bar() { return 5; }
};
```

gives the syntax error, "Error: Storage class 'dynamic' is not allowed here." But, the following code compiles.

```
#include <clxvcl.h>
#include <stdio.h>
class __declspec(delphiclass) func1 : public TObject {
public:
func1() {}
int virtual virtbar() { return 5; }
int __declspec(dynamic) dynbar() { return 5; }
};
class __declspec(delphiclass) func2 : public func1 {
public:
func2() {}
};
class __declspec(delphiclass) func3 : public func2 {
public:
func3() {}
int virtbar() { return 10; }
int dynbar() { return 10; }
};
int main()
{
func3 * Func3 = new func3;
func1 * Func1 = Func3;
printf("func3->dynbar: %d\n", Func3->dynbar());
printf("func3->virtbar: %d\n", Func3->virtbar());
printf("func1->dynbar: %d\n", Func1->dynbar());
printf("func1->virtbar: %d\n", Func1->virtbar());
delete Func3;
func2 * Func2 = new func2;
printf("func2->dynbar: %d\n", Func2->dynbar());
printf("func2->virtbar: %d\n", Func2->virtbar());
delete Func2;
```

```
return 0;
}
```

Dynamic attribute is inherited

Since dynamic functions are just like virtual functions, the dynamic attribute is automatically inherited. You can verify this by running the above example. When you generate assembly output with "bcc32 -S" you can examine the virtual tables of func1, func2, and func3, and you'll see how func2 has NO entry for dynbar, but it does have an entry for virtbar. Still, you can call dynbar in the func2 object:

Output:

```
func3->dynbar: 10
```

```
func3->virtbar: 10
```

```
func1->dynbar: 10
```

```
func1->virtbar: 10
```

```
func2->dynbar: 5
```

```
func2->virtbar: 5
```

Dynamic functions cannot be made virtual, and vice-versa

You cannot redeclare a virtual function to be dynamic; likewise, you cannot redeclare a dynamic function to be virtual. The following example gives errors:

```
#include <clxvcl.h>
```

```
#include <stdio.h>
```

```
class __declspec(delphiclass) fool : public TObject {
```

```
public:
```

```
fool() {}
```

```
int virtual virtbar() { return 5; }
```

```
int __declspec(dynamic) dynbar() { return 5; }
```

```
};
```

```
class __declspec(delphiclass) foo2 : public fool {
```

```
public:
```

```
foo2() {}
```

```
int __declspec(dynamic) virtbar() { return 10; }
```

```
int virtual dynbar() { return 10; }
```

```
};
```

```
Error : Cannot override a virtual with a dynamic function
```

```
Error : Cannot override a dynamic with a virtual function
```


Abstract Classes

This section contains Abstract Class topics.

In This Section

[Abstract Classes](#)

Abstract Classes

An abstract class is a class with at least one pure virtual function. A virtual function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {          // abstract class
    point center;
    .
    .
    .
public:
    where() { return center; }
    move(point p) { center = p; draw(); }
    virtual void rotate(int) = 0; // pure virtual function
    virtual void draw() = 0;      // pure virtual function
    virtual void hilite() = 0;    // pure virtual function
    .
    .
    .
    .
}
shape x; // ERROR: attempt to create an object of an abstract class
shape* sptr; // pointer to abstract class is OK
shape f(); // ERROR: abstract class cannot be a return type
int g(shape s); // ERROR: abstract class cannot be a function argument type
shape& h(shape&); // reference to abstract class as return
// value or function argument is OK
```

Suppose that D is a derived class with the abstract class B as its immediate base class. Then for each pure virtual function pvf in B, if D doesn't provide a definition for pvf, pvf becomes a pure member function of D, and D will also be an abstract class.

For example, using the class shape previously outlined,

```
class circle : public shape { // circle derived from abstract class
    int radius; // private
public:
    void rotate(int) { } // virtual function defined: no action
                        // to rotate a circle
    void draw();        // circle::draw must be defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a runtime error.

C++ Scope

This section contains C++ Scope topics.

In This Section

[C++ Scope](#)

[Class Scope](#)

[Hiding](#)

[C++ Scoping Rules Summary](#)

C++ Scope

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

Class Scope

The name M of a member of a class X has class scope “local to X”; it can be used only in the following situations:

- In member functions of X
- In expressions such as x.M, where x is an object of X
- In expressions such as xptr->M, where xptr is a pointer to an object of X
- In expressions such as X::M or D::M, where D is a derived class of X
- In forward references within the class of which it is a member

Names of functions declared as friends of X are not members of X; their names simply have enclosing scope.

Hiding

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: X::M. A hidden file scope (global) name can be referenced with the unary operator :: (for example, ::g). A class name X can be hidden by the name of an object, function, or enumerator declared within the scope of X, regardless of the order in which the names are declared. However, the hidden class name X can still be accessed by prefixing X with the appropriate keyword: class, struct, or union.

The point of declaration for a name x is immediately after its complete declaration but before its initializer, if one exists.

C++ Scoping Rules Summary

The following rules apply to all names, including typedef names and class names, provided that C++ allows such names in the particular context discussed:

- The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
- If no access control errors occur, the type of the object, function, class, typedef, and so on, is tested.
- If the name is used outside any function and class, or is prefixed by the unary scope access operator ::, and if the name is not qualified by the binary :: operator or the member selection operators . and ->, then the name must be a global object, function, or enumerator.

- If the name `n` appears in any of the forms `X::n`, `x.n` (where `x` is an object of `X` or a reference to `X`), or `ptr->n` (where `ptr` is a pointer to `X`), then `n` is the name of a member of `X` or the member of a class from which `X` is derived.
- Any name that hasn't been discussed yet and that is used in a static member function must either be declared in the block it occurs in or in an enclosing block, or be a global name. The declaration of a local name `n` hides declarations of `n` in enclosing blocks and global declarations of `n`. Names in different scopes are not overloaded.
- Any name that hasn't been discussed yet and that is used in a nonstatic member function of class `X` must either be declared in the block it occurs in or in an enclosing block, be a member of class `X` or a base class of `X`, or be a global name. The declaration of a local name `n` hides declarations of `n` in enclosing blocks, members of the function's class, and global declarations of `n`. The declaration of a member name hides declarations of the same name in base classes.
- The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a nondefining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
- A constructor initializer is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

Templates

This section contains Template topics.

In This Section

[Using Templates](#)

[template](#)

[Template Body Parsing](#)

Using Templates

Templates, also called generics or parameterized types, let you construct a family of related functions or classes. These topics introduce the basic concept of templates:

Exporting and importing templates

Template Body Parsing

Function Templates

Class Templates

Implicit and Explicit Template Functions

Template Compiler Switches

template

Category

C++-Specific Keywords

Syntax

```
template-declaration:templateclass
    template < template-argument-list > declaration
template-argument-list:
    template-argument
    template-argument-list, template argument
template-argument:
    type-argument
    argument-declaration
type-argument:
    class typename identifier
template-class-name:
    template-name < template-arg-list >
template-arg-list:
    template-arg
    template-arg-list , template-arg
template-arg:
    expression
    type-name
< template-argument-list > declaration
```

Description

Use templates (also called generics or parameterized types) to construct a family of related functions or classes.

Template Body Parsing

Earlier versions of the compiler didn't check the syntax of a template body unless the template was instantiated. A template body is now parsed immediately when seen like every other declaration.

```
template <class T> class X : T
{
    Int j; // Error: Type name expected in template X<T>
};
```

Let's assume that `Int` hasn't yet been defined. This means that `Int` must be a member of the template argument `T`. But it also might just be a typing error and should be `int` instead of `Int`. Because the compiler can't guess the right meaning it issues an error message.

If you want to access types defined by a template argument you should use a typedef to make your intention clear to the compiler:

```
template <class T> class X : T
{
    typedef typename T::Int Int;
    Int j;
};
```

You cannot just write

```
typedef T::Int;
```

as in earlier versions of the compiler. Not giving the typedef name was acceptable, but this now causes an error message.

All other templates mentioned inside the template body are declared or defined at that point. Therefore, the following example is ill-formed and will not compile:

```
template <class T> class X
{
    void f(NotYetDefinedTemplate<T> x);
};
```

All template definitions must end with a semicolon. Earlier versions of the compiler did not complain if the semicolon was missing.

Function Templates Overview

This section contains Function Templates Overview topics.

In This Section

[Function Templates](#)

[Overriding A Template Function](#)

[Implicit And Explicit Template Functions](#)

Function Templates

Consider a function `max(x, y)` that returns the larger of its two arguments. `x` and `y` can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters `x` and `y` to be declared at compile time. Without using templates, many overloaded versions of `max` are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the `#define` circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of `max(x, y)` is to compare compatible types. Unfortunately, using the macro allows a comparison between an `int` and a `struct`, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be. By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

```
template <class T> T max(T x, T y) {  
    return (x > y) ? x : y;  
};
```

The data type is represented by the template argument `<class T>`. When used in an application, the compiler generates the appropriate code for the `max` function according to the data type actually used in the call:

```
int i;  
Myclass a, b;  
int j = max(i,0);           // arguments are integers  
Myclass m = max(a,b);       // arguments are type Myclass
```

Any data type (not just a class) can be used for `<class T>`. The compiler takes care of calling the appropriate `operator>()`, so you can use `max` with arguments of any type for which `operator>()` is defined.

Overriding A Template Function

The previous example is called a function template (or generic function, if you like). A specific instantiation of a function template is called a template function. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>  
char *max(char *x, char *y){
```

```

    return(strcmp(x,y) > 0) ? x : y;
}

```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

Implicit And Explicit Template Functions

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```

template<class T> T max(T a, T b){
    return  (a > b) ? a : b;
}
void f(int i, char c)
{
    max(i, i);      // calls max(int ,int )
    max(c, c);      // calls max(char,char)
    max(i, c);      // no match for max(int,char)
    max(c, i);      // no match for max(char,int)
}

```

This code results in the following error messages:

```

Could not find a match for 'max(int,char)' in function f(int,char)Could not find a match for
'max(char,int)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)

```

If the user explicitly declares a function, this function, on the other hand, will participate fully in overload resolution. See the example of explicit function.

When searching for an exact match for template function parameters, trivial conversions are considered to be exact matches. See the example on trivial conversions.

Template functions with derived class pointer or reference arguments are permitted to match their public base classes. See the example of base class referencing.

Class Templates Overview

This section contains Class Templates Overview topics.

In This Section

[Class Templates](#)

[Template Arguments](#)

[Using Angle Brackets In Templates](#)

[Using Type-safe Generic Lists In Templates](#)

[Eliminating Pointers In Templates](#)

Class Templates

A class template (also called a generic class or class generator) lets you define a pattern for class definitions. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a type parameter to the class, the system will generate type-safe class definitions on the fly.

As with function templates, an explicit template class specialization can be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol Vector must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

Template Arguments

Multiple arguments are allowed as part of the class template declaration. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Both non-type template arguments such as size and type arguments can have default values. The value supplied for a non-type template argument must be a constant expression:

```
const int N = 128;
int i = 256;
Buffer<int, 2*N> b1; // OK
Buffer<float, i> b2; // Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

Using Angle Brackets In Templates

Be careful when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the > between x and 100 would prematurely close the template argument list.

Using Type-safe Generic Lists In Templates

In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

```
class GList
{
public:
    void insert( void * );
    void *peek();
    .
    .
    .
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public GList {
public:
    void insert( Foo *f ) { GList::insert( f ); }
    Foo *peek() { return (Foo *)GList::peek(); }
    .
    .
    .
};
```

This is type-safe. insert will only take arguments of type pointer-to-Foo or object-derived-from-Foo, so the underlying container will only hold pointers that in fact point to something of type Foo. This means that the cast in FooList::peek() is always safe, and you've created a true FooList. Now, to do the same thing for a BarList, a BazList, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and type-safety, you can once again use templates. See the example for type-safe generic list class. The C++ Standard Template Library (STL) has a rich set of type-safe collection classes.

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no runtime overhead imposed by this type safety.

Eliminating Pointers In Templates

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of virtual function calls required, since the compiler knows the actual types of the objects. This is beneficial if the virtual functions are small enough to be effectively inlined. It's difficult to inline virtual functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

```
template <class T> aBase {
    .
    .
    .
private:
    T buffer;
};
```

```
class anObject : public aSubject, public aBase<aFilebuf> {  
    .  
    .  
    .  
};
```

All the functions in aBase can call functions defined in aFilebuf directly, without having to go through a pointer. And if any of the functions in aFilebuf can be inlined, you'll get a speed improvement, because templates allow them to be inlined.

Compiler Template Switches

This section contains Compiler Template Switch topics.

In This Section

[Template Compiler Switches](#)

Template Compiler Switches

The -Jg family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler.

For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which instances will be user-defined so that they aren't generated from the template.

Template Generation Semantics

The C++ compiler generates the following methods for template instances:

- Those methods which were actually used
- Virtual methods of an instance
- All methods of explicitly instantiated classes

The advantage of this behavior is that it results in significantly smaller object files, libraries and executable files, depending on how heavily you use templates.

Optionally, you can use the '-Ja' switch to generate all methods.

You can also force all of the out-of-line methods of a template instance to be generated by using the explicit template instantiation syntax defined in the ISO/ANSI C++ Standard. The syntax is:

```
template class classname<template parameter>;
```

The following STL example directs the compiler to generate all out-of-line methods for the "list<char>" class, regardless of whether they are referenced by the user's code:

```
template class list<char>
```

You can also explicitly instantiate a single method, or a single static data member of a template class, which means that the method is generated to the .OBJ even though it is not used:

```
template void classname <template parameter>:: methodname ();
```


Exporting And Importing Templates

This section contains Exporting And Importing Template topics.

In This Section

[Exporting And Importing Templates](#)

Exporting And Importing Templates

The declaration of a template function or template class needs to be sufficiently flexible to allow it to be used in either a dynamic link library (shared library) or an executable file. The same template declaration should be available as an import and/or export, or without a modifier. To be completely flexible, the header file template declarations should not use `__export` or `__import` modifiers. This allows the user to apply the appropriate modifier at the point of instantiation depending on how the instantiation is to be used.

The following steps demonstrate exporting and importing of templates. The source code is organized in three files. Using the header file, code is generated in the dynamic link library.

1. Exportable/Importable Template Declarations

The header file contains all template class and template function declarations. An export/import version of the templates can be instantiated by defining the appropriate macro at compile time.

2. Compiling Exportable Templates

Write the source code for a dynamic link library. When compiled, this library has reusable export code for templates.

3. Using ImportTemplates

Now you can write a calling function that uses templates. This executable file is linked to the dynamic link library. Only objects that are not declared in the header file and which are instantiated in the main function cause the compiler to generate new code. Code for a newly instantiated object is written into `main.obj` file.

The Preprocessor

This section contains Preprocessor topics.

In This Section

- [Preprocessor Directives](#)
- [Defining And Undefined Macros](#)
- [Macros With Parameters Overview](#)
- [File Inclusion With #include](#)
- [Conditional Compilation Overview](#)
- [The #line Control Directive](#)
- [The #error Control Directive](#)
- [Pragma Directives Overview](#)
- [Predefined Macros Overview](#)

Preprocessor Directives

This section contains Preprocessor Directive topics.

In This Section

[Preprocessor Directives](#)

[# \(null Directive\)](#)

Preprocessor Directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them. The preprocessor supports these directives:

# (null directive)	#ifdef
#define	#ifndef
#elif	#undef
#else	#include
#endif	#line
#error	#pragma
#if	#import

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

(null Directive)

Syntax

```
#
```

Description

The null directive consists of a line containing the single character #. This line is always ignored.

Defining And Undefined Macros

This section contains Defining And Undefined Macro topics.

In This Section

[#define](#)

[#undef](#)

[Using The -D And -U Command-line Options](#)

[Keywords And Protected Words In Macros](#)

#define

Syntax

```
#define macro_identifier <token_sequence>
```

Description

The `#define` directive defines a macro. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Each occurrence of `macro_identifier` in your source code following this control line will be replaced in the original position with the possibly empty `token_sequence` (there are some exceptions, which are noted later). Such replacements are known as macro expansions. The token sequence is sometimes called the body of the macro.

An empty token sequence results in the removal of each affected macro identifier from the source code.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of nested macros: The expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, the directive will not be recognized by the preprocessor. There are these restrictions to macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro won't be expanded during its own expansion (so `#define A A` won't expand indefinitely).

Example

```
#define HI "Have a nice day!"  
#define empty
```

#undef

Syntax

```
#undef macro_identifier
```

Description

You can undefine a macro using the `#undef` directive. `#undef` detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined. No macro expansion occurs within `#undef` lines.

The state of being defined or undefined turns out to be an important property of an identifier, regardless of the actual definition. The `#ifdef` and `#ifndef` conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with `#define`, using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
#define BLOCK_SIZE 512
```

The middle line is bypassed if `BLOCK_SIZE` is currently defined; if `BLOCK_SIZE` is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

Using The -D And -U Command-line Options

Identifiers can be defined and undefined using the command-line compiler options `-D` and `-U`.

The command line

```
BCC32 -Ddebug=1; paradox=0; X -Umysym myprog.cbc++ -Ddebug=1:paradox=0:X -Umysym myprog.c
bc++ -Ddebug=1:paradox=0:X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
```

in the program.

Keywords And Protected Words In Macros

It is legal but ill-advised to use C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long    /* legal and possibly useful */
```

The following predefined global identifiers cannot appear immediately following a `#define` or `#undef` directive:

- `__DATE__` `__FILE__` `__LINE__`
- `__STDC__` `__TIME__`

Macros With Parameters Overview

This section contains Macros With Parameters Overview topics.

In This Section

- [Macros With Parameters](#)
- [Nesting Parentheses And Commas](#)
- [Token Pasting With ##](#)
- [Converting To Strings With #](#)
- [Using The Backslash \(\\) For Line Continuation](#)
- [Side Effects And Other Dangers](#)

Macros With Parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note there can be no whitespace between the macro identifier and the (. The optional `arg_list` is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a formal argument or placeholder.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional `actual_arg_list` must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal `arg_list` of the `#define` line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the `actual_arg_list`.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Nesting Parentheses And Commas

The `actual_arg_list` can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.

```
#define ERRMSG(x, str) showerr("Error:, x. str")
#define SUM(x,y) ((x) + (y))
```

Token Pasting With

You can paste (or merge) two tokens together by separating them with `##` (plus optional whitespace on either side). The preprocessor removes the whitespace and the `##`, combining the separate tokens into one new token. You can use this to construct identifiers.

Given the definition

```
#define VAR(i, j) (i##j)
```

the call `VAR(x, 6)` expands to `(x6)`. This replaces the older nonportable method of using `(i/**/j)`.

Converting To Strings With

The `#` symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement.

Given the following definition:

```
#define TRACE(flag) printf(#flag "=%d\n", flag)
```

the code fragment

```
int highval = 1024;
TRACE(highval);
```

becomes

```
int highval = 1024;
printf("highval" "=%d\n", highval);
```

which, in turn, is treated as

```
int highval = 1024;
printf("highval=%d\n", highval);
```

Using The Backslash (\) For Line Continuation

A long token sequence can straddle a line by using a backslash (`\`). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.

```
#define WARN "This is really a single-\
line warning."
```

Side Effects And Other Dangers

The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once.

Compare `CUBE` and `cube` in the following example.


```
int cube(int x) {  
    return x* x*x;  
}
```


File Inclusion With #include

This section contains File Inclusion With #include topics.

In This Section

[#include](#)

[Header File Search With <header_name>](#)

[Header File Search With "header_name"](#)

#include

Syntax

```
#include <header_name>
#include "header_name"
```

Description

The #include directive pulls in other named files, known as include files, header files, or headers, into the source code. The syntax has three versions:

- The first and second versions imply that no macro expansion will be attempted; in other words, header_name is never scanned for macro identifiers. header_name must be a valid file name with an extension (traditionally .h for header) and optional path name and path delimiters.
- The third version assumes that neither < nor " appears as the first non-whitespace character following #include; further, it assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <header_name> or "header_name" formats.

The preprocessor removes the #include line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the #include can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the header_name, only that directory will be searched.

The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file.

Header File Search With <header_name>

The <header_name> version specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

Header File Search With "header_name"

The "header_name" version specifies a user-supplied include file; the file is searched in the following order:

1. - in the same directory of the file that contains the #include statement
2. - in the directories of files that include (#include) that file
3. - the current directory
4. - along the path specified by the /I compiler option

Conditional Compilation Overview

This section contains Conditional Compilation Overview topics.

In This Section

[Conditional Compilation](#)

[Defined](#)

[#if, #elif, #else, And #endif](#)

[#ifdef And #ifndef](#)

Conditional Compilation

The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the #if, #ifdef, #ifndef, #else, #elif, and #endif directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

Defined

Syntax

```
#if defined([<identifier>])  
#elif defined([<identifier>])
```

Description

Use the defined operator to test if an identifier was previously defined using #define. The defined operator is only valid in #if and #elif expressions.

Defined evaluates to 1 (true) if a previously defined symbol has not been undefined (using #undef); otherwise, it evaluates to 0 (false).

Defined performs the same function as #ifdef.

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use defined repeatedly in a complex expression following the #if directive; for example,

```
#if defined(mysym) && !defined(yoursym)
```

#if, #elif, #else, And #endif

Syntax

```
#if constant-expression-1  
<section-1>  
<#elif constant-expression-2 newline section-2>  
.
```

```

.
.
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif

```

Description

The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The conditional directives `#if`, `#elif`, `#else`, and `#endif` work like the normal C conditional operators. If the constant-expression-1 (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by section-1, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the compiler. Otherwise, if constant-expression-1 evaluates to zero (false), section-1 is ignored (no macro expansion and no compilation).

In the true case, after section-1 has been preprocessed, control passes to the matching `#endif` (which ends this conditional sequence) and continues with next-section. In the false case, control passes to the next `#elif` line (if any) where constant-expression-2 is evaluated. If true, section-2 is processed, after which control moves on to the matching `#endif`. Otherwise, if constant-expression-2 is false, control passes to the next `#elif`, and so on, until either `#else` or `#endif` is reached. The optional `#else` is used as an alternative condition for which all previous tests have proved false. The `#endif` ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each `#if` must be matched with a closing `#endif`.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each `#if` can be matched with its correct `#endif`.

The constant expressions to be tested must evaluate to a constant integral value.

#ifdef And #ifndef

Syntax

```

#ifdef identifier
#ifndef identifier

```

Description

The `#ifdef` and `#ifndef` conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous `#define` command has been processed for that identifier and is still in force. The line

```

#ifdef identifier

```

has exactly the same effect as

```

#if 1

```

if identifier is currently defined, and the same effect as

```

#if 0

```

if identifier is currently undefined.

`#ifndef` tests true for the "not-defined" condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if identifier is currently defined, and the same effect as

```
#if 1
```

if identifier is currently undefined.

The syntax thereafter follows that of the `#if`, `#elif`, `#else`, and `#endif`.

An identifier defined as `NULL` is considered to be defined.

The #line Control Directive

This section contains #line Control Directive topics.

In This Section

[#line](#)

#line

Syntax

```
#line integer_constant <"filename">
```

Description

You can use the #line directive to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program.

The #line directive indicates that the following source line originally came from line number integer_constant of filename. Once the filename has been registered, subsequent #line commands relating to that file can omit the explicit filename argument.

Macros are expanded in #line arguments as they are in the #include directive.

The #line directive is primarily used by utilities that produce C code as output, and not in human-written code.

The #error Control Directive

This section contains #error Control Directive topics.

In This Section

[#error](#)

#error

Syntax

```
#error errmsg
```

Description

The #error directive generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional statement that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an #error directive within a conditional statement that is true for the undesired case.

Pragma Directives Overview

This section contains Pragma Directives Overview topics.

In This Section

- [#pragma Summary](#)
- [#pragma Anon_struct](#)
- [#pragma Argsused](#)
- [#pragma Codeseg](#)
- [#pragma Comment](#)
- [#pragma exit and #pragma startup](#)
- [#pragma Hdrfile](#)
- [#pragma Hdrstop](#)
- [#pragma Inline](#)
- [#pragma Intrinsic](#)
- [#pragma Link](#)
- [#pragma Message](#)
- [#pragma Pack](#)
- [#pragma Package](#)
- [#pragma Obsolete](#)
- [#pragma Option](#)
- [#pragma Resource](#)
- [#pragma Warn](#)

#pragma Summary

Syntax

```
#pragma directive-name
```

Description

With #pragma, you can set compiler directives in your source code, without interfering with other compilers that also support #pragma. If the compiler doesn't recognize directive-name, it ignores the #pragma directive without any error or warning message.

The Borland C++ supports the following #pragma directives:

- #pragma alignment
- #pragma anon_struct
- #pragma argsused
- #pragma checkoption
- #pragma codeseg
- #pragma comment
- #pragma defineonoption
- #pragma exit
- #pragma hdrfile
- #pragma hdrstop
- #pragma inline

- #pragma intrinsic
- #pragma link
- #pragma message
- #pragma nopushoptwarn
- #pragma obsolete
- #pragma option
- #pragma pack
- #pragma package
- #pragma resource
- #pragma startup
- #pragma undefineoption
- #pragma warn

#pragma Anon_struct

Syntax

```
#pragma anon_struct on
#pragma anon_struct off
```

Description

The anon_struct directive allows you to compile anonymous structures embedded in classes.

```
#pragma anon_struct on
```

```
struct S {
    int i;
    struct { // Embedded anonymous struct
        int j ;
        float x ;
    };
    class { // Embedded anonymous class
    public:
        long double ld;
    };
    S() { i = 1; j = 2; x = 3.3; ld = 12345.5;}
};
#pragma anon_struct off
```

```
void main()
```

```
{
    S mystruct;
    mystruct.x = 1.2; // Assign to embedded data.
}
```

#pragma Argsused

Syntax

```
#pragma argsused
```

Description

The argsused pragma is allowed only between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

#pragma Codeseg

Syntax

```
#pragma codeseg <seg_name> <"seg_class"> <group>
```

Description

The codeseg directive lets you name the segment, class, or group where functions are allocated. If the pragma is used without any of its options, the default code segment is used for function allocation.

#pragma Comment

Syntax

```
#pragma comment (comment type, "string")
```

Description

The comment directive lets you write a comment record into an output file. The comment type can be one of the following values:

Value	Explanation
exestr	The linker writes string into an .obj file. Your specified string is placed in the executable file. Such a string is never loaded into memory but can be found in the executable file by use of a suitable file search utility.
lib	Writes a comment record into an .obj file. A library module that is not specified in the linker's response-file can be specified by the comment LIB directive. The linker includes the library module name specified in string as the last library. Multiple modules can be named and linked in the order in which they are named.
user	The compiler writes string into the .obj file. The linker ignores the specified string.

#pragma exit and #pragma startup

Syntax

```
#pragma startup function-name <priority>  
#pragma exit function-name <priority>
```

Description

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the main function is called), or program exit (just before the program terminates through `_exit`).

The specified function-name must be a previously declared function taking no arguments and returning void; in other words, it should be declared as:

```
void func(void);
```

The optional priority parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100.

Warning: Do not use priority values less than 64. Priorities from 0 to 63 are reserved for ISO startup and shutdown mechanisms.

#pragma Hdrfile

Syntax

```
#pragma hdrfile "filename.csm"
```

Description

This directive sets the name of the file in which to store precompiled headers.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option `-H=filename` or Use Precompiled Headers to change the name of the file used to store precompiled headers.

#pragma Hdrstop

Syntax

```
#pragma hdrstop
```

Description

This directive terminates the list of header files eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers.

Precompiled header files can be shared between the source files of your project only if the `#include` directives before `#pragma hdrstop` are identical. Therefore, you get the best compiler performance if you include common header files of your project before the `#pragma hdrstop`, and specific ones after it. Make sure the `#include` directives before the `#pragma hdrstop` are identical in all the source files, or that there are only very few variations.

The integrated development environment generates code to enhance precompiled header performance. For example, after a New Application, source file "Unit1.cpp" will look like this (comments added):

```
#include <clxvcl.h>           // common header file
#pragma hdrstop              // terminate list here
```

Use this pragma directive only in source files. The pragma has no effect when it is used in a header file.

#pragma Inline

Syntax


```
#pragma inline
```

Description

This directive is equivalent to the -B command-line compiler option.

This is best placed at the top of the file, because the compiler restarts itself with the -B option when it encounters #pragma inline.

#pragma Intrinsic

Syntax

```
#pragma intrinsic [-]function-name
```

Description

Use #pragma intrinsic to override command-line switches or project options to control the inlining of functions.

When inlining an intrinsic function, always include a prototype for that function before using it.

#pragma Link

Syntax

```
#pragma link "[path]modulename[.ext]"
```

Description

The directive instructs the linker to link the file into an executable file.

By default, the linker searches for modulename in the local directory and any path specified by the -L option. You can use the path argument to specify a directory.

By default, the linker assumes a .obj extension.

#pragma Message

Syntax

```
#pragma message ("text" ["text"["text" ...]])  
#pragma message text
```

Description

Use #pragma message to specify a user-defined message within your program code.

The first form requires that the text consist of one or more string constants, and the message must be enclosed in parentheses. (This form is compatible with Microsoft C.) This form will output the constant contained between the double quotes regardless of whether it is a macro or not.

The second form uses the text following the #pragma for the text of the warning message. With this form of the #pragma, any macro references are expanded before the message is displayed.

The third form will output the macro-expanded value of text following the #pragma, if it is #defined. If it is not #defined, you'll get an ill-formed pragma warning.

User-defined messages are displayed as messages, not warnings.

Display of user-defined messages is on by default and can be turned on or off with the Show Messages option. This option corresponds to the compiler's -wmsg switch.

Messages are only displayed in the IDE if **Show general messages** is checked on the Compiling tab under **Project** ▶ **Options** ▶ **Compiler**.

#pragma Pack

Syntax

```
#pragma pack([push | pop][,]] [identifier[,]] [n])
```

Description

The #pragma pack(n).directive is the same as using the #pragma optionspecifically with the -a compiler option. n is the byte alignment that determines how the compiler aligns data in stored memory. For more information see the -a compiler option. #pragma pack can also be used with push and pop arguments, which provide the same functionality as the #pragma option directive using push and pop. The following table compares the use of #pragma pack with #pragma option.

#pragma pack	#pragma option
#pragma pack(n)	#pragma option -an
#pragma pack(push, n)	#pragma option push -an
#pragma pack(pop)	#pragma option pop

The #pragma pack directive also supports an identifier argument which must be used in combination with either push or pop.

#pragma pack with no arguments

```
#pragma pack()
```

Using #pragma pack with no arguments will set the packing size to the starting -aX alignment (which defaults to 8). The starting -aX alignment is considered the alignment at the start of the compile AFTER all command-line options have been processed.

#pragma pack using a value for n

```
#pragma pack(8)
```

Using #pragma pack with a value for 'n' will set the current alignment to 'n'. Valid alignments for 'n' are: 1,2,4,8, and 16.

#pragma pack using push

```
#pragma pack(push)
```

Using #pragma pack with push will push the current alignment on an internal stack of alignments.

#pragma pack using push, identifier

```
#pragma pack(push, ident)
```

Using #pragma pack with push and an identifier will associate the pushed alignment with 'identifier'.

#pragma pack using push and n

```
#pragma pack(push, 8)
#pragma pack(push, ident, 8)
```

Using `#pragma pack` with `push` with a value for 'n', will execute `pragma pack push` or `pragma pack push identifier`, and afterwards set the current alignment to 'n'.

#pragma pack using pop

```
#pragma pack(pop)
```

Using `#pragma pack` with `pop` will pop the alignment stack and set the alignment to the last alignment pushed. If the `pop` does not find a corresponding push, the entire stack of alignments is popped, and a warning is issued, and the alignment reverts to the starting -aX alignment..

#pragma pack using pop, identifier

```
#pragma pop(pop, ident)
```

Using `#pragma pack` with `pop` and an identifier will pop the stack until the identifier is found and set the alignment to the alignment pushed by the previous corresponding `#pragma pack(push, identifier)`. If the `pop` with identifier does not find the corresponding push with an identifier, the entire stack of alignments is popped, and a warning is issued to the user:

```
W8083: Pragma pack pop with no matching pack push
```

The alignment will then be reset to the starting -aX alignment.

#pragma pack using pop and n

```
#pragma pack(pop, 8)
#pragma pack(pop, ident, 8)
```

Using `#pragma pack` with `pop` and a value for 'n', will execute `pragma pack pop` or `pragma pack pop identifier`. Afterwards the current alignment is set to 'n', unless `pop` fails to find a corresponding push, in which case 'n' is ignored, a warning is issued, and the alignment reverts to the starting -aX alignment.

Error conditions

Specifying an 'identifier' without push or pop is an error.

Specifying an alignment different from 1,2,4,8,16 is an error.

Warning conditions

Using `#pragma pop` without a corresponding push issues a warning.

#pragma Package

Syntax

```
#pragma package(smart_init)
#pragma package(smart_init, weak)
```

Description: smart_init argument

The `#pragma package(smart_init)` assures that packaged units are initialized in the order determined by their dependencies. (Included by default in package source file.) Typically, you would use the `#pragma package` for `.cpp` files that are built as packages.

This pragma affects the order of initialization of that compilation unit. For units, initialization occurs in the following order:

- 1. By their "uses" dependencies, that is, if unitA depends on unitB, unitB must be initialized before unitA.
- 2. The link order.
- 3. Priority order within the unit.

For regular object files (those not built as units), initialization happens first according to priority order and then link order. Changing the link order of the object files changes the order in which the global object constructors get called.

The following examples show how the initialization differs between units and regular object files.

Take as an example three unit files, A, B and C that are "smart initialized" with `#pragma package(smart_init)` and have priority values (defined by the priority parameter of the `#pragma startup`) set of 10, 20 and 30. The functions are named according to their priority value and the parent object file, so the names are a10, a20, a30, b10, and so on.

Since all three are units, and if A uses B and C and the link order is A, B then C, the order of initialization is:

B10 B20 B30 C10 C20 C30 A10 A20 A30

If the above were object files, not units, the order would be:

A10 B10 C10 A20 B20 C20 A30 B30 C30

The `.cpp` files that use `#pragma package(smart_init)` also require that any `#pragma link` references to other object files from a `.cpp` file that declares `#pragma package(smart_init)`, must be resolved by a unit. `#pragma link` references to non object files can still be resolved by libraries, etc.

Description: weak packages

The `#pragma package(smart_init, weak)` directive affects the way an object file is stored in a package's `.bpi` and `.bpl` files. If `#pragma package(smart_init, weak)` appears in a unit file, the compiler omits the unit from BPLs when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged".

`#pragma package(smart_init, weak)` is used to eliminate conflicts among packages that may depend on the same external library.

Unit files containing the `#pragma package(smart_init, weak)` directive must not have global variables.

For more information about using weak packages, see Weak packaging.

#pragma Obsolete

Syntax

```
#pragma obsolete identifier
```

Description

Use `#pragma obsolete` to give a warning the first time the preprocessor encounters identifier in your program code after the pragma. The warning states that the identifier is obsolete.

#pragma Option

Syntax

```
#pragma option options
#pragma option push options
#pragma option pop
#pragma nopushoptwarn
```

Description

Use `#pragma option` to include command-line options within your program code. `#pragma option` can also be used with the `push` or `pop` arguments.

`#pragma option [options...]`

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. For example, both of the following are valid:

```
#pragma option -C
#pragma option -C -A
```

Any of the toggle options (such as `-a` or `-K`) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, and then return it to its default, without having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a `pragma option` include:

B	c	dname
Dname=string	efilename	E
Fx	h	Ifilename
lexset	M	o
P	Q	S
T	Uname	V
X	Y	

- You can use `#pragmas`, `#includes`, `#define`, and some `#ifs` in the following cases:
- Before the use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an `#if`, `#ifdef`, `#ifndef` or `#elif` directive.
- Before the occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can appear only in a `#pragma option` command before these events. These options are:

Efilename	f	i#
m*	npath	ofilename
U	W	z
<hr/> *		

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

1	h	r
2	k	rd
A	N	v
Ff	O	y
G	p	Z

The following options can be changed at any time and take effect immediately:

A	gn	zE
b	jn	zF
C	K	zH
d	wxxx	

The options can appear followed by a dot (.) to reset the option to its command-line state.

#pragma option using push or pop

The #pragma option directive can also be used with the push and pop arguments to enable you to easily modify compiler directives.

Using the #pragma option push allows you to save all (or a selected subset of) options before including any files that could potentially change many compiler options and warnings, and then, with the single statement, #pragma option pop, return to the previous state. For example:

```
#pragma option push
#include <theworld.h>
#pragma option pop
#include "mystuff.h"
```

The `#pragma option push` directive first pushes all compiler options and warning settings on a stack and then handles any options (if supplied). The following examples show how the `#pragma option push` can be used with or without options:

```
#pragma option push -C -A
#pragma option push
```

The `#pragma option pop` directive changes compiler options and warnings by popping the last set of options and warnings from the stack. It gives a warning, "Pragma option pop with no matching option push", if the stack is empty, in which case nothing happens.

The following generates a warning about an empty stack:

```
#pragma option push
#pragma option pop
#pragma option pop      /* Warning */
```

We do not recommend it, but you can turn off this warning with the directive: `#pragma warn -nop`.

If you try to specify any options after pop, you get the error, "Nothing allowed after pragma option pop." For example, the following produces an error:

```
#pragma option pop -C/* ERROR */
```

If your stack of pushed options is not the same at the start of a file as at the end of a file, you receive a warning: "Previous options and warnings not restored." To turn off this warning, use the directive, `#pragma nopushoptwarn`.

#pragma Resource

Syntax

```
#pragma resource "*.xdfm"
```

Description

This pragma causes the file to be marked as a form unit and requires matching .dfm and header files. All such files are managed by the IDE.

If your form requires any variables, they must be declared immediately after the pragma resource is used. The declarations must be of the form

```
TFormName *Formname;
```

#pragma Warn

Syntax

```
#pragma warn [+|-|.]xxx
```

Description

The warn pragma lets you override specific -wxxx command-line options or check Display Warnings. xxx is the three-letter or four-digit message identifier used by the command-line option.

Predefined Macros Overview

This section contains Predefined Macros Overview topics.

In This Section

[Predefined Macros](#)

Predefined Macros

The compiler predefines certain global identifiers, known as manifest constants. Most global identifiers begin and end with two underscores (__).

Note: For readability, underscores are often separated by a single blank space. In your source code, you should never insert whitespace between underscores.

Macro	Value	Description
__BCOPT__	1	Defined in any compiler that has an optimizer.
__BCPLUSPLUS__	0x0570	Defined if you've selected C++ compilation; will increase in later releases.
__BORLANDC__	0x0570	Version number.
__CDECL__	1	Defined if Calling Convention is set to cdecl; otherwise undefined.
__CHAR_UNSIGNED	1	Defined by default indicating that the default char is unsigned char. Use the -K compiler option to undefine this macro.
__CODEGUARD__		Defined whenever one of the CodeGuard compiler options is used; otherwise it is undefined.
__CONSOLE__	1	When defined, the macro indicates that the program is a console application.
__CPPUNWIND	1	Enable stack unwinding. This is true by default; use -xd-!ALink (OSCGExceptions1) to disable.
__cplusplus	1	Defined if in C++ mode; otherwise, undefined.
__DATE__	String literal	Date when processing began on the current file.
__DLL__	1	Defined whenever the -WD compiler option is used; otherwise it is undefined.
__FILE__	String literal	Name of the current file being processed.
__FLAT__	1	Defined when compiling in 32-bit flat memory model.
__FUNC__	String literal	Name of the current function being processed. More details.
__LINE__	Decimal constant	Number of the current source file line being processed.
__M_I86	0x12c	Always defined. The default value is 300. You can change the value to 400 or 500 by using the /4 or /5 compiler options.
__MT__	1	Defined only if the -tWM option is used. It specifies that the multithread library is to be linked.
__PASCAL__	1	Defined if Calling Convention is set to Pascal; otherwise undefined.
__STDC__	1	Defined if you compile with the -A compiler option; otherwise, it is undefined.
__TCPLUSPLUS__	0x0570	Version number.
__TEMPLATES__	1	Defined as 1 for C++ files (meaning that templates are supported); otherwise, it is undefined.
__TIME__	String literal	Time when processing began on the current file.

<code>__TLS__</code>	1	Thread Local Storage. Always true.
<code>__TURBOC__</code>	0x0570	Will increase in later releases.
<code>_WCHAR_T</code>		Defined only for C++ programs to indicate that <code>wchar_t</code> is an intrinsically defined data type.
<code>_WCHAR_T_DEFINED</code>		Defined only for C++ programs to indicate that <code>wchar_t</code> is an intrinsically defined data type.
<code>_Windows</code>		Defined when compiling on the Windows platform.
<code>__WIN32__</code>	1	Defined for console and GUI applications on the Windows platform.
<code>__linux__</code>		Defined when compiling on the Linux platform.

Note: `__DATE__`, `__FILE__`, `__FUNC__`, `__LINE__`, `__STDC__`, and `__TIME__` cannot be redefined or undefined.

Keywords, By Category

This section contains Keywords, By Category topics.

In This Section

[C++ Specific Keywords](#)

[C++ Builder Keyword Extensions](#)

[Modifiers](#)

[Operators](#)

[Special Types](#)

[Statements](#)

[Storage Class Specifiers](#)

[Type Specifiers](#)

C++ Specific Keywords

This section contains C++ Specific Keyword topics.

In This Section

[Asm, _asm, __asm](#)
[Bool, False, True](#)
[Catch](#)
[Class](#)
[const_cast \(typeid Operator\)](#)
[delete](#)
[dynamic_cast \(typeid Operator\)](#)
[Explicit](#)
[Friend](#)
[Inline](#)
[Mutable](#)
[namespace](#)
[new](#)
[Operator](#)
[Private](#)
[Protected](#)
[Public](#)
[reinterpret_cast \(typeid Operator\)](#)
[__rtti, -RT Option](#)
[static_cast \(typeid Operator\)](#)
[template](#)
[This](#)
[Throw](#)
[Try](#)
[Typeid](#)
[Typename](#)
[using \(declaration\)](#)
[Virtual](#)
[Wchar_t](#)

Asm, _asm, __asm

Category

Keyword extensions, C++-Specific Keywords

Syntax

```
asm <opcode> <operands> <; or newline>_asm _asm  
_asm <opcode> <operands> <; or newline>  
__asm <opcode> <operands> <; or newline>
```

Description

Use the `asm`, `_asm`, or `__asm` keyword to place assembly language statements in the middle of your C or C++ source code. Any C++ symbols are replaced by the appropriate assembly language equivalents.

You can group assembly language statements by beginning the block of statements with the `asm` keyword, then surrounding the statements with braces (`{}`).

Bool, False, True

Category

C++-Specific Keywords

Syntax

```
bool <identifier>;
```

Description

Use `bool` and the literals `false` and `true` to perform Boolean logic tests.

The `bool` keyword represents a type that can take only the value `false` or `true`. The keywords `false` and `true` are Boolean literals with predefined values. `false` is numerically zero and `true` is numerically one. These Boolean literals are rvalues; you cannot make an assignment to them.

You can convert an rvalue that is of type `bool` to an rvalue that is `int` type. The numerical conversion sets `false` to zero and `true` becomes one.

You can convert arithmetic, enumeration, pointer, or pointer to member rvalue types to an rvalue of type `bool`. A zero value, null pointer value, or null member pointer value is converted to `false`. Any other value is converted to `true`.

Catch

Category

Statements, C++-Specific Keywords

Syntax

```
catch (exception-declaration) compound-statement
```

Description

The exception handler is indicated by the `catch` keyword. The handler must be used immediately after the statements marked by the `try` keyword. The keyword `catch` can also occur immediately after another `catch`. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

Class

Category

C++-Specific Keywords, Type specifiers

Syntax

```
<classkey> <classname> <baselist> { <member list> }
```

- `<classkey>` is either a class, struct, or union.
- `<classname>` can be any name unique within its scope.
- `<baselist>` lists the base class(es) that this class derives from. `<baselist>` is optional

- <member list> declares the class's data members and member functions.

Description

Use the class keyword to define a C++ class.

Within a class:

- the data are called data members
- the functions are called member functions

const_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
const_cast< T > (arg)
```

Description

Use the const_cast operator to add or remove the const or volatile modifier from a type.

In the statement, const_cast< T > (arg), T and arg must be of the same type except for const and volatile modifiers. The cast is resolved at compile time. The result is of type T. Any number of const or volatile modifiers can be added or removed with a single const_cast expression.

A pointer to const can be converted to a pointer to non-const that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A const object or a reference to const cast results in a non-const object or reference that is otherwise an identical type.

The const_cast operator performs similar typecasts on the volatile modifier. A pointer to volatile object can be cast to a pointer to non-volatile object without otherwise changing the type of the object. The result is a pointer to the original object. A volatile-type object or a reference to volatile-type can be converted into an identical non-volatile type.

delete

Category

Operators, C++-Specific Keywords

Syntax

```
void operator delete(void *ptr) throw();  
void operator delete(void *ptr, const std::nothrow_t&) throw();  
void operator delete[](void *ptr) throw();  
void operator delete[](void *ptr, const std::nothrow_t &) throw();  
void operator delete(void *ptr, void *) throw(); // Placement form  
void operator delete[](void *ptr, void *) throw(); // Placement form
```

Description

The delete operator deallocates a memory block allocated by a previous call to new. It is similar but superior to the standard library function free.

You should use the delete operator to remove all memory that has been allocated by the new operator. Failure to free memory can result in memory leaks.

The default placement forms of operator delete are reserved and cannot be redefined. The default placement delete operator performs no action (since no memory was allocated by the default placement new operator). If you overload the placement version of operator new, it is a good idea (though not strictly required) to provide the overload the placement delete operator with the corresponding signature.

dynamic_cast (typeid Operator)

Category

C++-Specific Keywords

Description

In the expression, `dynamic_cast< T > (ptr)`, T must be a pointer or a reference to a defined class type or void*. The argument ptr must be an expression that resolves to a pointer or reference.

If T is void* then ptr must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if T is a pointer and ptr is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If T is a reference and ptr is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at runtime.

If successful, `dynamic_cast< T > (ptr)` converts ptr to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the `Bad_cast` exception is thrown.

Note: Runtime type identification (RTTI) is required for `dynamic_cast`.

Explicit

Category

C++-Specific Keywords

Syntax

```
explicit <single-parameter constructor declaration>
```

Description

Normally, a class with a single-parameter constructor can be assigned a value that matches the constructor type. This value is automatically (implicitly) converted into an object of the class type to which it is being assigned. You can prevent this kind of implicit conversion from occurring by declaring the constructor of the class with the explicit keyword. Then all objects of that class must be assigned values that are of the class type; all other assignments result in a compiler error.

Objects of the following class can be assigned values that match the constructor type or the class type:

```
class X {  
public:
```



```
X(int);
X(const char*, int = 0);
};
```

Then, the following assignment statements are legal.

```
void f(X arg) {
    X a = 1;
    X B = "Jessie";
    a = 2;
}
```

However, objects of the following class can be assigned values that match the class type only:

```
class X {
public:
    explicit X(int);
    explicit X(const char*, int = 0);
};
```

The explicit constructors then require the values in the following assignment statements to be converted to the class type to which they are being assigned.

```
void f(X arg) {
    X a = X(1);
    X b = X("Jessie",0);
    a = X(2);
}
```

Friend

Category

C++-Specific Keywords

Syntax

```
friend <identifier>;
```

Description

Use friend to declare a function or class with full access rights to the private and protected members of the class, without being a member of that class. The outside class has full access to the class that declares that outside class a friend.

In all other respects, the friend is a normal function in terms of scope, declarations, and definitions.

Inline

Category

C++-Specific Keywords

Syntax

```
inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the inline keyword to declare or define C++ inline functions.

Inline functions are best reserved for small, frequently used functions.

Mutable

Category

C++-Specific Keywords, Storage class specifiers

Syntax

```
mutable <variable name>;
```

Description

Use the mutable specifier to make a variable modifiable even though it is in a const-qualified expression.

Using the mutable Keyword

Only class data members can be declared mutable. The mutable keyword cannot be used on static or const names. The purpose of mutable is to specify which data members can be modified by const member functions. Normally, a const member function cannot modify data members.

namespace

Category

C++-Specific Keywords

Description

Most real-world applications consist of more than one source file. The files can be authored and maintained by more than one developer. Eventually, the separate files are organized and linked to produce the final application. Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as function or class body, or translation unit) must share the same global namespace. Therefore, multiple definitions of names discovered while linking separate modules require some way to distinguish each name. The solution to the problem of name clashes in the global scope is provided by the C++ namespace mechanism.

The namespace mechanism allows an application to be partitioned into a number of subsystems. Each subsystem can define and operate within its own scope. Each developer is free to introduce whatever identifiers are convenient within a subsystem without worrying about whether such identifiers are being used by someone else. The subsystem scope is known throughout the application by a unique identifier.

It only takes two steps to use C++ namespaces. The first is to uniquely identify a namespace with the keyword namespace. The second is to access the elements of an identified namespace by applying the using keyword.

new

Category

Operators, C++-Specific Keywords

Syntax

```

void *operator new(std::size_t size) throw(std::bad_alloc);
void *operator new(std::size_t size, const std::nothrow_t &) throw();
void *operator new[](std::size_t size) throw(std::bad_alloc);
void *operator new[](std::size_t size, const std::nothrow_t &) throw();
void *operator new(std::size_t size, void *ptr) throw();    // Placement form
void *operator new[](std::size_t size, void *ptr) throw(); // Placement form

```

Description

The new operators offer dynamic storage allocation, similar but superior to the standard library function malloc. These allocation functions attempt to allocate size bytes of storage. If successful, new returns a pointer to the allocated memory. If the allocation fails, the new operator will call the new_handler function. The default behavior of new_handler is to throw an exception of type bad_alloc. If you do not want an exception to be thrown, use the nothrow version of operator new. The nothrow versions return a null pointer result on failure, instead of throwing an exception.

The default placement forms of operator new are reserved and cannot be redefined. You can, however, overload the placement form with a different signature (i.e. one having a different number, or different type of arguments). The default placement forms accept a pointer of type void, and perform no action other than to return that pointer, unchanged. This can be useful when you want to allocate an object at a known address. Using the placement form of new can be tricky, as you must remember to explicitly call the destructor for your object, and then free the pre-allocated memory buffer. Do not call the delete operator on an object allocated with the placement new operator.

A request for non-array allocation uses the appropriate operator new() function. Any request for array allocation will call the appropriate operator new[]() function. Remember to use the array form of operator delete[](), when deallocating an array created with operator new[]().

Note: Arrays of classes require that a default constructor be defined in the class.

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

Operator

Category

Operators, C++-Specific Keywords

Syntax

```

operator <operator symbol>( <parameters> )
{
    <statements>;
}

```

Description

Use the operator keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as *this is implicitly the first argument.

When you overload an operator as a friend, you can specify two arguments.

Private

Category

C++-Specific Keywords

Syntax

```
private: <declarations>
```

Description

Access to private class members is restricted to member functions within the class, and to friend classes.

Class members are private by default.

Structure (struct) and union members are public by default. You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

Protected

Category

C++-Specific Keywords

Syntax

```
protected: <declarations>
```

Description

Access to protected class members is restricted to member functions within the class, member functions of derived classes, and to friend classes.

Structure (struct) and union members are public by default. You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

Public

Category

C++-Specific Keywords

Syntax

```
public: <declarations>
```

Description

A public class member can be accessed by any function.

Members of a struct or union are public by default.

You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

reinterpret_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
reinterpret_cast< T > (arg)
```

Description

In the statement, `reinterpret_cast< T > (arg)`, `T` must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral `arg` can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

__rtti, -RT Option

Category (__rtti keyword)

Modifiers, C++ Keyword Extensions, C++-Specific Keywords

Description

Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the `-RT-` option to disable it or `-RT` to enable it.

If RTTI is disabled, or if the argument to `typeid` is a pointer or a reference to a non-polymorphic class, `typeid` returns a reference to a `const type_info` object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic runtime type identification (where appropriate) by using the `__rtti` keyword in the class definition.

When runtime type identification is disabled, if any base class is declared `__rtti`, then all polymorphic base classes must also be declared `__rtti`.

```
struct __rtti S1 { virtual s1func(); }; /* Polymorphic */
struct __rtti S2 { virtual s2func(); }; /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism, type information might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations without RTTI, you should declare `X` with the `__rtti` modifier. Otherwise, switching the order of the base classes for the class `X` results in the compile-time error "Can't inherit non-RTTI class from RTTI base 'S1'."

```
struct __rtti S1 { virtual func(); }; /* Polymorphic class */
struct S2 { }; /* Non-polymorphic class */
struct __rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with `__rtti`. This makes it safe to mix the order and type of classes.

In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared `__rtti`.

```
struct __rtti S1 { }; // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { }; // The order is not essential
```

Neither the `__rtti` keyword, nor enabling RTTI will make a static class into a polymorphic class.

static_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
static_cast< T > (arg)
```

Description

In the statement, `static_cast< T > (arg)`, T must be a pointer, reference, arithmetic type, or enum type. Both T and arg must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using `static_cast` achieves exactly the same thing.

Integral types can be converted to enum types. A request to convert arg to a value that is not an element of enum is undefined.

The null pointer is converted to the null pointer value of the destination type, T.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class X to a pointer to some class Y if X is a base class for Y. A static conversion can be made only under the following conditions:

- if an unambiguous conversion exists from Y to X
- if X is not a virtual base class

An object can be explicitly converted to a reference type X& if a pointer to that object can be explicitly converted to an X*. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.

An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.

A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When T is a reference the result of `static_cast< T > (arg)` is an lvalue. The result of a pointer or reference cast refers to the original expression.

template

Category

C++-Specific Keywords

Syntax

```
template-declaration:templateclass
    template < template-argument-list > declaration
template-argument-list:
    template-argument
    template-argument-list, template argument
template-argument:
    type-argument
    argument-declaration
type-argument:
    class typename identifier
template-class-name:
    template-name < template-arg-list >
template-arg-list:
    template-arg
    template-arg-list , template-arg
template-arg:
    expression
    type-name
< template-argument-list > declaration
```

Description

Use templates (also called generics or parameterized types) to construct a family of related functions or classes.

This

Category

C++-Specific Keywords

Syntax

```
class X {intpublicintthis
int a;
public:
X (int b) {this -> a = b;}
};
```

Description

In nonstatic member functions, the keyword `this` is a pointer to the object for which the function is called. All calls to nonstatic member functions pass `this` as a hidden argument.

`this` is a local variable available in the body of any nonstatic member function. Use it implicitly within the function for member references. It does not need to be declared and it is rarely referred to explicitly in a function definition.

For example, in the call `x.func(y)`, where `y` is a member of `X`, the keyword `this` is set to `&x` and `y` is set to `this->y`, which is equivalent to `x.y`.

Static member functions do not have a `this` pointer because they are called with no particular object in mind. Thus, a static member function cannot access nonstatic members without explicitly specifying an object with `.` or `->`.

Throw

Category

Statements, C++-Specific Keywords

Syntax

```
throw assignment-expression
```

Description

When an exception occurs, the throw expression initializes a temporary object of the type T (to match the type of argument arg) used in throw(T arg). Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

Try

Category

Statements, C++-Specific Keywords

Syntax

```
try compound-statement handler-list
```

Description

The try keyword is supported only in C++ programs. Use __try in C programs. C++ also allows __try.

A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is transferred to the handler

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

Typeid

Category

Operators, C++-Specific Keywords

Syntax

```
typeid( expression )typeid  
typeid( type-name )
```

Description

You can use typeid to get runtime identification of types and expressions. A call to typeid returns a reference to an object of type const type_info. The returned object represents the type of the typeid operand.

If the typeid operand is a dereferenced pointer or a reference to a polymorphic type, typeid returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, typeid returns an object that represents the static type.

You can use the typeid operator with fundamental data types as well as user-defined types.

When the typeid operand is a Delphi class object/reference, typeid returns the static rather than runtime type.

If the typeid operand is a dereferenced NULL pointer, the Bad_typeid exception is thrown.

Typename

Category

C++-Specific Keywords

Syntax 1

```
typename identifier
```

Syntax 2

```
template < typename identifier > class identifier
```

Description

Use the syntax 1 to reference a type that you have not yet defined. See example 1.

Use syntax 2 in place of the class keyword in a template declaration. See example 2.

Note: When using the typename keyword with templates, the compiler will not always report an error in cases where the ANSI standard requires the typename keyword. The compiler will flag the omission of typename when the compiler is invoked with the -A switch. For example, given the following code:

```
#include <stdio.h>
struct A{ typedef int AInt; };
```

Note: The compiler will flag the omission of typename when the compiler is invoked with the -A switch.

Note: Compile with: bcc32 (no -A switch)

```
bc++bcc32 test.cpp
```

The result is no error. The Compiler should not assume AInt is a typename, but it does unless -A switch is used

Note: Compile with: bcc32 (-A switch)

```
bc++bcc32 -A test.cpp
```

The result is:

Error E2089 47071.cpp 7: Identifier 'AInt' cannot have a type qualifier

Error E2303 47071.cpp 7: Type name expected

Error E2139 47071.cpp 7: Declaration missing ;

Both results are as expected.

using (declaration)

Category

C++-Specific Keywords

Description

You can access namespace members individually with the using-declaration syntax. When you make a using declaration, you add the declared identifier to the local namespace. The grammar is

using-declaration:

```
using :: unqualified-identifier;
```

Virtual

Category

C++-Specific Keywords

Syntax

```
virtual class-namevirtual  
virtual function-name
```

Description

Use the virtual keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as virtual, you can redefine it in any derived class, even if the number and type of arguments are the same.

The redefined function overrides the base class function.

Wchar_t

Category

C++-Specific Keywords, Type specifiers

Syntax

```
wchar_t <identifier>;
```

Description

In C++ programs, wchar_t is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In Borland C++, A wchar_t type is the same size, signedness, and alignment requirement as an unsigned short type.

C++ Builder Keyword Extensions

This section contains C++ Builder Keyword Extension topics.

In This Section

[Asm, _asm, __asm](#)
[__automated](#)
[Cdecl, cdecl, __cdecl](#)
[__classid](#)
[__closure](#)
[__declspec](#)
[__declspec\(dllexport\)](#)
[__declspec\(dllimport\)](#)
[__declspec\(naked\)](#)
[__declspec\(noreturn\)](#)
[__declspec\(nothrow\)](#)
[__declspec\(novtable\)](#)
[__declspec\(property\)](#)
[__declspec\(selectany\)](#)
[__declspec\(thread\)](#)
[__declspec\(uuid\("ComObjectGUID"\)\)](#)
[__except](#)
[__export, __export](#)
[__fastcall, __fastcall](#)
[__finally](#)
[__import, __import](#)
[__inline](#)
[__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types](#)
[__msfastcall](#)
[__msreturn](#)
[__thread, Multithread Variables](#)
[Pascal, __pascal, __pascal](#)
[__property](#)
[__published](#)
[__rtti, -RT Option](#)
[__stdcall, __stdcall](#)
[__try](#)

Asm, _asm, __asm

Category

Keyword extensions, C++-Specific Keywords

Syntax

```
asm <opcode> <operands> <; or newline>_asm __asm
_asm <opcode> <operands> <; or newline>
__asm <opcode> <operands> <; or newline>
```

Description

Use the `asm`, `_asm`, or `__asm` keyword to place assembly language statements in the middle of your C or C++ source code. Any C++ symbols are replaced by the appropriate assembly language equivalents.

You can group assembly language statements by beginning the block of statements with the `asm` keyword, then surrounding the statements with braces (`{}`).

`__automated`

Category

Keyword extensions

Syntax

```
__automated: <declarations>
```

Description

The visibility rules for automated members are identical to those of public members. The only difference between automated and public members is that OLE automation information is generated for member functions and properties that are declared in an automated section. This OLE automation type information makes it possible to create OLE Automation servers.

- For a member function, the types of all member function parameters and the function result (if any) must be automatable. Likewise, for a property, the property type and the types of any array property parameters must be automatable. The automatable types are: `Currency`, `OleVariant`, `DelphiInterface`, `double`, `int`, `float`, `short`, `String`, `TDateTime`, `Variant`, and `unsigned short`. Declaring member functions or properties that use non-automatable types in an `__automated` section results in a compile-time error.
- Member function declarations must use the `__fastcall` calling convention.
- Member functions can be virtual.
- Member functions may add `__dispid(constant int expression)` after the closing parenthesis of the parameter list.
- Property declarations can only include access specifiers (`__dispid`, `read`, and `write`). No other specifiers (`index`, `stored`, `default`, `nodefault`) are allowed.
- Property access specifiers must list a member function identifier. Data member identifiers are not allowed.
- Property access member functions must use the `__fastcall` calling convention.
- Property overrides (property declarations that don't include the property type) are not allowed.

`Cdecl`, `_cdecl`, `__cdecl`

Category

Modifiers, Keyword extensions

Syntax

```
cdecl <data/function definition> ;_cdecl__cdecl  
_cdecl <data/function definition> ;  
__cdecl <data/function definition> ;
```

Description

Use a `cdecl`, `_cdecl`, or `__cdecl` modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use `cdecl`, `_cdecl`, or `__cdecl` in front of a function,

it affects how the parameters are passed (parameters are pushed right to left, and the caller cleans up the stack). The `__cdecl` modifier overrides the compiler directives and IDE options.

The `cdecl`, `_cdecl`, and `__cdecl` keywords are specific to Borland C++.

`__classid`

Category

Operators, Keyword extensions

Syntax

```
__classid(classType)
```

Description

The `__classid` operator was added to support the VCL framework. Normally, programmers should not directly use this operator. For more information, see the keyword extensions.

`__closure`

Category

Keyword extensions

Syntax

```
<type> ( __closure * <id> ) (<param list>);
```

Description

The keyword `__closure` was added to support the VCL framework and is used when declaring event handler functions. For more information, see the keyword extensions.

`__declspec`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec(<decl-modifier>)
```

Description

For a list of `__declspec` keyword arguments used for the VCL framework, see CLXVCL class declarations.

Use the `__declspec` keyword to indicate the storage class attributes for a variable or function.

The `__declspec` keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The `__declspec` keyword and its argument can appear anywhere in the declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);           // illegal
void __export f(void)           // correct
```

```
void __declspec(dllexport) f(void);           // correct
__declspec(dllexport) void f(void);          // correct
class __declspec(dllexport) ClassName { }    // correct
```

In addition to the CLX-related arguments listed above, the supported decl-modifier arguments are:

- `dllexport`
- `dllimport`
- `naked`
- `noreturn`
- `nothrow`
- `novtable`
- `property`
- `selectany`
- `thread`
- `uuid`

These arguments are equivalent to the following storage class attribute keywords.

Argument	Comparable keyword
<code>dllexport</code>	<code>__export</code>
<code>dllimport</code>	<code>__import</code>
<code>thread</code>	<code>__thread</code>

`__declspec(dllexport)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllexport ) declarator
```

The `dllexport` storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to export functions, data, and objects from a DLL. This attribute explicitly defines the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as `dllexport` eliminates the need for a module-definition (.DEF) file, at least with respect to the specification of exported functions.

Note: `dllexport` replaces the `__export` keyword.

`__declspec(dllimport)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllimport ) declarator
```

The `dllimport` storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to import functions, data, and objects to a DLL.

Note: Note: `dllimport` replaces the `__import` keyword.

__declspec(naked)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( naked ) declarator
```

Use of the `naked` argument suppresses the prolog/epilog code. Be aware when using `__declspec(naked)` that it does not set up a normal stack frame. A function with `__declspec(naked)` will not preserve the register values that are normally preserved. It is the programmer's responsibility to conform to whatever conventions the caller of the function expects.

You can use this feature to write your own prolog/epilog code using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

The `naked` attribute is relevant only to the definition of a function and is not a type modifier.

Example

This code defines a function with the `naked` attribute:

```
// Example of the naked attribute
__declspec( naked ) int func( formal_parameters )
{
    // Function body
}
```

__declspec(noreturn)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( noreturn ) declarator
```

This `__declspec` attribute tells the compiler that a function does not return. As a consequence, the compiler knows that the code following a call to a `__declspec(noreturn)` function is unreachable.

If the compiler finds a function with a control path that does not return a value, it generates a warning. If the control path cannot be reached due to a function that never returns, you can use `__declspec(noreturn)` to prevent this warning or error.

Example

Consider the following code. The else clause does not contain a return statement, so the programmer declares fatal as `__declspec(noreturn)` to avoid an error or warning message.

```
__declspec(noreturn) extern void fatal ()
{
    // Code omitted
}
int foo()
{
    if(...)
        return 1;
    else if(...)
        return 0;
    else
        fatal();
}
```

`__declspec(nothrow)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( nothrow ) declarator
```

This is a `__declspec` extended attribute that can be used in the declaration of functions. This attribute tells the compiler that the declared function and the functions it calls never throw an exception. With the synchronous exception handling model, now the default, the compiler can eliminate the mechanics of tracking the lifetime of certain unwindable objects in such a function, and significantly reduce the code size.

The following three declarations are equivalent:

```
#define WINAPI __declspec(nothrow) __stdcall
```

```
void WINAPI foo1();
```

```
void __declspec(nothrow) __stdcall foo2();
```

```
void __stdcall foo3() throw();
```

Using `void __declspec(nothrow) __stdcall foo2();` has the advantage that you can use an API definition, such as the illustrated by the `#define` statement, to easily specify `nothrow` on a set of functions. The third declaration, `void __stdcall foo3() throw();` is the syntax defined by the C++ standard.

`__declspec(novtable)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax


```
__declspec( novtable ) declarator
```

This form of `__declspec` can be applied to any class declaration, but should only be applied to pure interface classes, that is classes that will never be instantiated on their own. The `__declspec` stops the compiler from generating code to initialize the `vfptr` in the constructor(s) and destructor of the class. In many cases, this removes the only references to the `vtable` that are associated with the class and, thus, the linker will remove it. Using this form of `__declspec` can result in a significant reduction in code size

`__declspec(property)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

This attribute can be applied to non-static “virtual data members” in a class or structure definition. The compiler treats these “virtual data members” as data members by changing their references into function calls.

When the compiler sees a data member declared with this attribute on the right of a member-selection operator (“.” or “->”), it converts the operation to a get or put function, depending on whether such an expression is an l-value or an r-value. In more complicated contexts, such as “+=”, a rewrite is performed by doing both get and put.

This attribute can also be used in the declaration of an empty array in a class or structure definition.

Example

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

The above statement indicates that `x[]` can be used with one or more array indices. In this case:

```
i=p->x[a][b]
```

will be turned into:

```
i=p->GetX(a, b),
```

and

```
p->x[a][b] = i
```

will be turned into

```
p->PutX(a, b, i);
```

`__declspec(selectany)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( selectany ) declarator
```

A global data item can normally be initialized only once in an application or library. This attribute can be used in initializing global data defined by headers, when the same header appears in more than one source file.

Note This attribute can only be applied to the actual initialization of global data items that are externally visible.

Example

This code shows how to use the selectany attribute:

```
//Correct - x1 is initialized and externally visible
```

```
__declspec(selectany) int x1=1;
```

```
//Incorrect - const is by default static in C++, so
```

```
//x2 is not visible externally (This is OK in C, since
```

```
//const is not by default static in C)
```

```
const __declspec(selectany) int x2 =2;
```

```
//Correct - x3 is extern const, so externally visible
```

```
extern const __declspec(selectany) int x3=3;
```

```
//Correct - x4 is extern const, so it is externally visible
```

```
extern const int x4;
```

```
const __declspec(selectany) int x4=4;
```

```
//Incorrect - __declspec(selectany) is applied to the uninitialized //declaration of x5 extern __declspec(selectany) int x5;
```

__declspec(thread)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( thread ) declarator
```

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process allocates storage for thread-specific data.

The thread extended storage-class modifier is used to declare a thread local variable. The thread attribute must be used with the __declspec keyword.

__declspec(uuid("ComObjectGUID"))

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( uuid("ComObjectGUID") ) declarator
```

The compiler attaches a GUID to a class or structure declared or defined (full COM object definitions only) with the `uuid` attribute. The `uuid` attribute takes a string as its argument. This string names a GUID in normal registry format with or without the { } delimiters. For example:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
```

```
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

This attribute can be applied in a redeclaration. This allows the system headers to supply the definitions of interfaces such as `IUnknown`, and the redeclaration in some other header (such as `COMDEF.H`) to supply the GUID.

The keyword `__uuidof` can be applied to retrieve the constant GUID attached to a user-defined type.

__except

Category

Statements, Keyword extensions

Syntax

```
__except (expression) compound-statement
```

Description

The `__except` keyword specifies the action that should be taken when the exception specified by `expression` has been raised.

_export, __export

Category

Modifiers, Keyword extensions

Form 1

```
class _export <class name>
```

Form 2

```
return_type _export <function name>
```

Form 3

```
data_type _export <data name>
```

Description

These modifiers are used to export classes, functions, and data.

The linker enters functions flagged with `_export` or `__export` into an export table for the module.

Using `_export` or `__export` eliminates the need for an `EXPORTS` section in your module definition file.

Functions that are not modified with `_export` or `__export` receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note: If you use `_export` or `__export` to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need a module definition file.

`_fastcall, __fastcall`

Category

Modifiers, Keyword extensions

Syntax

```
return-value _fastcall function-name(parm-list) __fastcall  
return-value __fastcall function-name(parm-list)
```

Description

Use the `__fastcall` modifier to declare functions that expect parameters to be passed in registers. The first three parameters are passed (from left to right) in EAX, EDI, and ECX, if they fit in the register. The registers are not used if the parameter is a floating-point or struct type.

All form class member functions must use the `__fastcall` convention.

The compiler treats this calling convention as a new language specifier, along the lines of `_cdecl` and `_pascal`.

Functions declared using `_cdecl` or `_pascal` cannot have the `_fastcall` modifier because they use the stack to pass parameters. Likewise, the `__fastcall` modifier cannot be used together with `_export`.

The compiler prefixes the `__fastcall` function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

For Microsoft C++ style `__fastcall` implementation, see `__msfastcall` and `__msreturn`.

Note: Note: The `__fastcall` modifier is subject to name mangling. See the description of the `-VC` option.

`__finally`

Category

Statements, Keyword extensions

Syntax

```
__finally {compound-statement}
```

Description

The `__finally` keyword specifies actions that should be taken regardless of how the flow within the preceding `__try` exits.

The following is the code fragment shows how to use the `try/__finally` construct:

```
#include <stdio.h>#include <string.h>#include <windows.h>class Exception{public:Exception  
(char* s = "Unknown"){what = strdup(s); }Exception(const Exception& e ){what = strdup  
(e.what); } ~Exception() {free(what); } char* msg() const  
{return what; }private:char* what;};int main(){float e, f, g;try {try {f = 1.0;  
g = 0.0;try {puts("Another exception.");e = f / g; }__except  
(EXCEPTION_EXECUTE_HANDLER) {puts("Caught a C-based exception.");throw(Exception
```

```

("Hardware error: Divide by 0"));      }    }catch(const Exception& e)    {printf("Caught C
++ Exception: %s :\n", e.msg());      }    }__finally {puts("C++ allows __finally too!"); }
return e;}
#include <string.h>
#include <windows.h>
class Exception
{
public:
Exception(char* s = "Unknown"){what = strdup(s);      }
Exception(const Exception& e ){what = strdup(e.what); }
~Exception()      {free(what);      }
char* msg() const      {return what;      }
private:
char* what;
};
int main()
{
float e, f, g;
try
{
try
{
f = 1.0;
g = 0.0;
try
{
puts("Another exception:");
e = f / g;
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
puts("Caught a C-based exception.");
throw(Exception("Hardware error: Divide by 0"));
}
}
catch(const Exception& e)
{
printf("Caught C++ Exception: %s :\n", e.msg());
}
}
__finally
{
puts("C++ allows __finally too!");
}
return e;
}

```

```

#include <iostream>#include <stdexcept>using namespace std;class MyException : public
exception {public:virtual const char *what() const throw() {return("MyException occurred.
");}}; // Give me any integer...void myFunc(int a){ MyException e; // ...but not that one.
if(a == 0) throw(e);}void main(){ int g; // Note __finally must be in its own try
block (with no preceding catch). try { try { g = 0; myFunc(g); }
catch(MyException &e) { cout << "Exception: " << e.what() << endl; } } __finally
{ cout << "Finally block reached." << endl; }
#include <stdexcept>
using namespace std;
class MyException : public exception {
public:
virtual const char *what() const throw() {
return("MyException occurred.");
}
}

```

```

}
};
// Give me any integer...
void myFunc(int a)
{
    MyException e;
    // ...but not that one.
    if(a == 0)
        throw(e);
}
void main()
{
    int g;
    // Note __finally must be in its own try block (with no preceding catch).
    try {
        try {
            g = 0;
            myFunc(g);
        }
        catch(MyException &e) {
            cout << "Exception: " << e.what() << endl;
        }
    }
    __finally {
        cout << "Finally block reached." << endl;
    }
}

```

Running the above program results in the following:

```

Another exception:Caught a C-based exception.Caught C++ exception[Hardware error: Divide by 0]C++ allows __finally too!Exception: MyException occurred.Finally block reached.
Caught a C-based exception.Caught C++ exception[Hardware error: Divide by 0]C++ allows __finally too!Exception: MyException occurred.Finally block reached.

```

`__import, __import`

Category

Modifiers, Keyword extensions

Form 1

```

class __import <class name>class __import
class __import <class name>

```

Form 2

```

return_type __import <function name>__import
return_type __import <function name>

```

Form 3

```

data_type __import <data name>__import
data_type __import <data name>

```

Description

This keyword can be used as a class, function, or data modifier.

__inline

Category

Keyword extensions

Syntax

```
__inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the __inline keyword to declare or define C or C++ inline functions. The behavior of the __inline keyword is identical to that of the inline keyword, which is only supported in C++.

Inline functions are best reserved for small, frequently used functions.

__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types

Category

Keyword extensions

Description

You can specify the size for integer types. You must use the appropriate suffix when using extended integers.

Type	Suffix	Example	Storage
__int8	i8	__int8 c = 127i8;	8 bits
__int16	i16	__int16 s = 32767i16;	16 bits
__int32	i32	__int32 i = 123456789i32;	32 bits
__int64	i64	__int64 big = 12345654321i64;	64 bits
unsigned __int64	ui64	unsigned __int64 hugeInt = 1234567887654321ui64;	64 bits

__msfastcall

Category

Modifiers, Keyword extensions

Syntax

```
__msfastcall <function-name>
```

Description

This calling convention emulates the Microsoft implementation of the fastcall calling conversion. The first two DWORD or smaller arguments are passed in ECX and EDX registers, all other arguments are passed from right to left. The called function is responsible for removing these arguments from the stack.

__msreturn

Category

Modifiers, Keyword extensions

Syntax

```
__msreturn <function-name>
```

Description

This calling convention is used for Microsoft compatible __fastcall calling convention return values. Structures with a size that is greater than 4 bytes and less than 9 bytes, and having at least one of its members sized 4 bytes or larger, are returned in EAX/EDX.

__thread, Multithread Variables

Category

Keyword extensions

Description

The keyword __thread is used in multithread programs to preserve a unique copy of global and static class variables. Each program thread maintains a private copy of a __thread variable for each thread.

The syntax is Type __thread variable__name. For example

```
int __thread x;
```

This statement declares an integer type variable that will be global but private to each thread in the program in which the statement occurs.

Pascal, _pascal, __pascal

Category

Modifiers, Keyword extensions

Syntax

```
pascal <data-definition/function-definition> ;_pascal__pascal  
_pascal <data-definition/function-definition> ;  
__pascal <data-definition/function-definition> ;
```

Description

Use the pascal, _pascal, and __pascal keywords to declare a variable or a function using a Pascal-style naming convention (the name is in uppercase).

In addition, pascal declares Delphi language-style parameter-passing conventions when applied to a function header (parameters pushed left or right; the called function cleans up the stack).

In C++ programs, functions declared with the pascal modifier will still be mangled.

__property

Category

Keyword extensions

Syntax

```
<property declaration> ::=  
    __property <type> <id> [ <prop dim list> ] = "{" <prop attrib list> "}"  
  
<prop dim list> ::=  
    "[" <type> [ <id> ] "]" [ <prop dim list> ]  
  
<prop attrib list> ::=  
    <prop attrib> [ , <prop attrib list> ]  
  
<prop attrib> ::=  
    read = <data/function id> |  
    write = <data/function id> |  
    stored = <data/function id> |  
    stored = <boolean constant> |  
    default = <constant> |  
    nodefault |  
    index = <const int expression>
```

Description

The __property keyword was added to support the VCL

__published

Category

Keyword extensions

Syntax

```
__published: <declarations>
```

Description

The __published keyword was added to support the VCL.

__rtti, -RT Option

Category (__rtti keyword)

Modifiers, C++ Keyword Extensions, C++-Specific Keywords

Description

Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the -RT- option to disable it or -RT to enable it.

If RTTI is disabled, or if the argument to typeid is a pointer or a reference to a non-polymorphic class, typeid returns a reference to a const type_info object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic runtime type identification (where appropriate) by using the __rtti keyword in the class definition.

When runtime type identification is disabled, if any base class is declared `__rtti`, then all polymorphic base classes must also be declared `__rtti`.

```
struct __rtti S1 { virtual s1func(); }; /* Polymorphic */
struct __rtti S2 { virtual s2func(); }; /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism, type information might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations without RTTI, you should declare X with the `__rtti` modifier. Otherwise, switching the order of the base classes for the class X results in the compile-time error "Can't inherit non-RTTI class from RTTI base 'S1'."

```
struct __rtti S1 { virtual func(); }; /* Polymorphic class */
struct S2 { }; /* Non-polymorphic class */
struct __rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with `__rtti`. This makes it safe to mix the order and type of classes.

In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared `__rtti`.

```
struct __rtti S1 { }; // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { }; // The order is not essential
```

Neither the `__rtti` keyword, nor enabling RTTI will make a static class into a polymorphic class.

`__stdcall`, `__stdcall`

Category

Modifiers, Keyword extensions

Syntax

```
__stdcall <function-name>__stdcall
__stdcall <function-name>
```

Description

The `__stdcall` and `__stdcall` keywords force the compiler to generate function calls using the Standard calling convention. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments.

Such functions comply with the standard WIN32 argument-passing convention.

Note: Note: The `__stdcall` modifier is subject to name mangling. See the description of the `-VC` option.

`__try`

Category

Statements, Keyword extensions

Syntax

```
__try compound-statement handler-list __try  
__try compound-statement termination-statement
```

Description

The `__try` keyword is supported in both C and C++ programs. You can also use `try` in C++ programs.

A block of code in which an exception can occur must be prefixed by the keyword `__try`. Following the `try` keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the normal program flow is interrupted. The program begins a search for a handler that matches the exception. If the exception is generated in a C module, it is possible to handle the structured exception in either a C module or a C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

If no handler is found, the program will call the `terminate` function. If no exceptions are thrown, the program executes in the normal fashion.

Modifiers

This section contains Modifier topics.

In This Section

[Cdecl, _cdecl, __cdecl](#)

[Const](#)

[__declspec](#)

[__declspec\(dllexport\)](#)

[__declspec\(dllimport\)](#)

[__declspec\(naked\)](#)

[__declspec\(noreturn\)](#)

[__declspec\(nothrow\)](#)

[__declspec\(novtable\)](#)

[__declspec\(property\)](#)

[__declspec\(selectany\)](#)

[__declspec\(thread\)](#)

[__declspec\(uuid\("ComObjectGUID"\)\)](#)

[__dispid](#)

[__export, __export](#)

[__fastcall, __fastcall](#)

[__import, __import](#)

[__msfastcall](#)

[__msreturn](#)

[Pascal, __pascal, __pascal](#)

[__rtti, -RT Option](#)

[__stdcall, __stdcall](#)

[Volatile](#)

Cdecl, _cdecl, __cdecl

Category

Modifiers, Keyword extensions

Syntax

```
cdecl <data/function definition> ;_cdecl__cdecl
_cdecl <data/function definition> ;
__cdecl <data/function definition> ;
```

Description

Use a cdecl, _cdecl, or __cdecl modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use cdecl, _cdecl, or __cdecl in front of a function, it affects how the parameters are passed (parameters are pushed right to left, and the caller cleans up the stack). The __cdecl modifier overrides the compiler directives and IDE options.

The cdecl, _cdecl, and __cdecl keywords are specific to Borland C++.

Const

Category

Modifiers

Syntax

```
const <variable name> [ = <value> ] ;constconst  
<function name> ( const <type>*<variable name> ;)  
<function name> const;
```

Description

Use the const modifier to make a variable value unmodifiable.

Use the const modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a const result in a compiler error.

A const pointer cannot be modified, though the object to which it points can be changed. Consider the following examples.

```
const float pi    = 3.14;  
const maxint  = 12345;    // When used by itself, const is equivalent to int.  
char *const str1 = "Hello, world";    // A constant pointer  
char const *str2 = "Borland Software Corporation"; // A pointer to a constant character  
string.
```

Given these declarations, the following statements are illegal.

```
pi    = 3.0;    // Assigns a value to a const.  
i      = maxint++; // Increments a const.  
str1 = "Hi, there!" // Points str1 to something else.
```

Using the const Keyword in C++ Programs

C++ extends const to include classes and member functions. In a C++ class definition, use the const modifier following a member function declaration. The member function is prevented from modifying any data in the class.

A class object defined with the const keyword attempts to use only member functions that are also defined with const. If you call a member function that is not defined as const, the compiler issues a warning that a non-const function is being called for a const object. Using the const keyword in this manner is a safety feature of C++.

Warning: A pointer can indirectly modify a const variable, as in the following:

```
*(int *)&my_age = 35;
```

If you use the const modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int printf (const char *format, ...);
```

printf is prevented from modifying the format string.

__declspec

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec(<decl-modifier>)
```

Description

For a list of __declspec keyword arguments used for the VCL framework, see CLXVCL class declarations.

Use the __declspec keyword to indicate the storage class attributes for a variable or function.

The __declspec keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The __declspec keyword and its argument can appear anywhere in the declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);           // illegal
void __export f(void)           // correct
void __declspec(dllexport) f(void); // correct
__declspec(dllexport) void f(void); // correct
class __declspec(dllexport) ClassName { } // correct
```

In addition to the CLX-related arguments listed above, the supported decl-modifier arguments are:

- dllexport
- dllimport
- naked
- noreturn
- nothrow
- novtable
- property
- selectany
- thread
- uuid

These arguments are equivalent to the following storage class attribute keywords.

Argument	Comparable keyword
dllexport	__export
dllimport	__import
thread	__thread

__declspec(dllexport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllexport ) declarator
```

The `dllexport` storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to export functions, data, and objects from a DLL. This attribute explicitly defines the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as `dllexport` eliminates the need for a module-definition (`.DEF`) file, at least with respect to the specification of exported functions.

Note: `dllexport` replaces the `__export` keyword.

`__declspec(dllexport)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllimport ) declarator
```

The `dllimport` storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to import functions, data, and objects to a DLL.

Note: Note: `dllimport` replaces the `__import` keyword.

`__declspec(naked)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( naked ) declarator
```

Use of the `naked` argument suppresses the prolog/epilog code. Be aware when using `__declspec(naked)` that it does not set up a normal stack frame. A function with `__declspec(naked)` will not preserve the register values that are normally preserved. It is the programmer's responsibility to conform to whatever conventions the caller of the function expects.

You can use this feature to write your own prolog/epilog code using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

The `naked` attribute is relevant only to the definition of a function and is not a type modifier.

Example

This code defines a function with the `naked` attribute:

```
// Example of the naked attribute
__declspec( naked ) int func( formal_parameters )
{
    // Function body
}
```

`__declspec(noreturn)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( noreturn ) declarator
```

This `__declspec` attribute tells the compiler that a function does not return. As a consequence, the compiler knows that the code following a call to a `__declspec(noreturn)` function is unreachable.

If the compiler finds a function with a control path that does not return a value, it generates a warning. If the control path cannot be reached due to a function that never returns, you can use `__declspec(noreturn)` to prevent this warning or error.

Example

Consider the following code. The `else` clause does not contain a `return` statement, so the programmer declares `fatal` as `__declspec(noreturn)` to avoid an error or warning message.

```
__declspec(noreturn) extern void fatal ()
{
    // Code omitted
}
int foo()
{
    if(...)
        return 1;
    else if(...)
        return 0;
    else
        fatal();
}
```

`__declspec(nothrow)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( nothrow ) declarator
```

This is a `__declspec` extended attribute that can be used in the declaration of functions. This attribute tells the compiler that the declared function and the functions it calls never throw an exception. With the synchronous exception handling model, now the default, the compiler can eliminate the mechanics of tracking the lifetime of certain unwindable objects in such a function, and significantly reduce the code size.

The following three declarations are equivalent:

```
#define WINAPI __declspec(nothrow) __stdcall
```

```
void WINAPI foo1();
```

```
void __declspec(nothrow) __stdcall foo2();
```

```
void __stdcall foo3() throw();
```

Using `void __declspec(nothrow) __stdcall foo2();` has the advantage that you can use an API definition, such as the illustrated by the `#define` statement, to easily specify `nothrow` on a set of functions. The third declaration, `void __stdcall foo3() throw();` is the syntax defined by the C++ standard.

__declspec(novtable)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( novtable ) declarator
```

This form of `__declspec` can be applied to any class declaration, but should only be applied to pure interface classes, that is classes that will never be instantiated on their own. The `__declspec` stops the compiler from generating code to initialize the `vfp`tr in the constructor(s) and destructor of the class. In many cases, this removes the only references to the `vtable` that are associated with the class and, thus, the linker will remove it. Using this form of `__declspec` can result in a significant reduction in code size

__declspec(property)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( property( get=get_func_name ) ) declarator  
__declspec( property( put=put_func_name ) ) declarator  
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

This attribute can be applied to non-static “virtual data members” in a class or structure definition. The compiler treats these “virtual data members” as data members by changing their references into function calls.

When the compiler sees a data member declared with this attribute on the right of a member-selection operator (“.” or “->”), it converts the operation to a get or put function, depending on whether such an expression is an l-value or an r-value. In more complicated contexts, such as “+=”, a rewrite is performed by doing both get and put.

This attribute can also be used in the declaration of an empty array in a class or structure definition.

Example

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

The above statement indicates that `x[]` can be used with one or more array indices. In this case:

```
i=p->x[a][b]
```

will be turned into:

```
i=p->GetX(a, b),
```

and

```
p->x[a][b] = i
```

will be turned into

```
p->PutX(a, b, i);
```

__declspec(selectany)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( selectany ) declarator
```

A global data item can normally be initialized only once in an application or library. This attribute can be used in initializing global data defined by headers, when the same header appears in more than one source file.

Note This attribute can only be applied to the actual initialization of global data items that are externally visible.

Example

This code shows how to use the selectany attribute:

```
//Correct - x1 is initialized and externally visible
```

```
__declspec(selectany) int x1=1;
```

```
//Incorrect - const is by default static in C++, so
```

```
//x2 is not visible externally (This is OK in C, since
```

```
//const is not by default static in C)
```

```
const __declspec(selectany) int x2 =2;
```

```
//Correct - x3 is extern const, so externally visible
```

```
extern const __declspec(selectany) int x3=3;
```

```
//Correct - x4 is extern const, so it is externally visible
```

```
extern const int x4;
```

```
const __declspec(selectany) int x4=4;
```

```
//Incorrect - __declspec(selectany) is applied to the uninitialized //declaration of x5 extern __declspec(selectany) int x5;
```

__declspec(thread)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( thread ) declarator
```

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process allocates storage for thread-specific data.

The thread extended storage-class modifier is used to declare a thread local variable. The thread attribute must be used with the `__declspec` keyword.

`__declspec(uuid("ComObjectGUID"))`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( uuid("ComObjectGUID") ) declarator
```

The compiler attaches a GUID to a class or structure declared or defined (full COM object definitions only) with the `uuid` attribute. The `uuid` attribute takes a string as its argument. This string names a GUID in normal registry format with or without the `{ }` delimiters. For example:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
```

```
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

This attribute can be applied in a redeclaration. This allows the system headers to supply the definitions of interfaces such as `IUnknown`, and the redeclaration in some other header (such as `COMDEF.H`) to supply the GUID.

The keyword `__uuidof` can be applied to retrieve the constant GUID attached to a user-defined type.

`__dispid`

Category

Modifiers

Syntax

```
__dispid(constant int expression)
```

Description

A member function that has been declared in the `__automated` section of a class can include an optional `__dispid` (constant int expression) directive. The directive must be declared after the closing parenthesis of the parameter list.

The constant int expression gives the Automation dispatch ID of the member function or property. If a `dispid` directive is not used, the compiler automatically picks a number one larger than the largest dispatch ID used by any member function or property in the class and its base classes.

Specifying an already-used dispatch ID in a `dispid` directive causes a compile-time error.

`__export, __export`

Category

Modifiers, Keyword extensions

Form 1

```
class _export <class name>
```

Form 2

```
return_type _export <function name>
```

Form 3

```
data_type _export <data name>
```

Description

These modifiers are used to export classes, functions, and data.

The linker enters functions flagged with `_export` or `__export` into an export table for the module.

Using `_export` or `__export` eliminates the need for an EXPORTS section in your module definition file.

Functions that are not modified with `_export` or `__export` receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note: If you use `_export` or `__export` to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need a module definition file.

`_fastcall`, `__fastcall`

Category

Modifiers, Keyword extensions

Syntax

```
return-value _fastcall function-name(parm-list) __fastcall  
return-value __fastcall function-name(parm-list)
```

Description

Use the `__fastcall` modifier to declare functions that expect parameters to be passed in registers. The first three parameters are passed (from left to right) in EAX, EDX, and ECX, if they fit in the register. The registers are not used if the parameter is a floating-point or struct type.

All form class member functions must use the `__fastcall` convention.

The compiler treats this calling convention as a new language specifier, along the lines of `_cdecl` and `_pascal`.

Functions declared using `_cdecl` or `_pascal` cannot have the `_fastcall` modifier because they use the stack to pass parameters. Likewise, the `__fastcall` modifier cannot be used together with `_export`.

The compiler prefixes the `__fastcall` function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

For Microsoft C++ style `__fastcall` implementation, see `__msfastcall` and `__msreturn`.

Note: Note: The `__fastcall` modifier is subject to name mangling. See the description of the `-VC` option.

`_import, __import`

Category

Modifiers, Keyword extensions

Form 1

```
class _import <class name>class __import  
class __import <class name>
```

Form 2

```
return_type _import <function name>__import  
return_type __import <function name>
```

Form 3

```
data_type _import <data name>__import  
data_type __import <data name>
```

Description

This keyword can be used as a class, function, or data modifier.

`__msfastcall`

Category

Modifiers, Keyword extensions

Syntax

```
__msfastcall <function-name>
```

Description

This calling convention emulates the Microsoft implementation of the fastcall calling conversion. The first two DWORD or smaller arguments are passed in ECX and EDX registers, all other arguments are passed from right to left. The called function is responsible for removing these arguments from the stack.

`__msreturn`

Category

Modifiers, Keyword extensions

Syntax

```
__msreturn <function-name>
```

Description

This calling convention is used for Microsoft compatible __fastcall calling convention return values. Structures with a size that is greater than 4 bytes and less than 9 bytes, and having at least one of its members sized 4 bytes or larger, are returned in EAX/EDX.

Pascal, _pascal, __pascal

Category

Modifiers, Keyword extensions

Syntax

```
pascal <data-definition/function-definition> ;_pascal__pascal  
_pascal <data-definition/function-definition> ;  
__pascal <data-definition/function-definition> ;
```

Description

Use the pascal, _pascal, and __pascal keywords to declare a variable or a function using a Pascal-style naming convention (the name is in uppercase).

In addition, pascal declares Delphi language-style parameter-passing conventions when applied to a function header (parameters pushed left or right; the called function cleans up the stack).

In C++ programs, functions declared with the pascal modifier will still be mangled.

__rtti, -RT Option

Category (__rtti keyword)

Modifiers, C++ Keyword Extensions, C++-Specific Keywords

Description

Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the -RT- option to disable it or -RT to enable it.

If RTTI is disabled, or if the argument to typeid is a pointer or a reference to a non-polymorphic class, typeid returns a reference to a const type_info object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic runtime type identification (where appropriate) by using the __rtti keyword in the class definition.

When runtime type identification is disabled, if any base class is declared __rtti, then all polymorphic base classes must also be declared __rtti.

```
struct __rtti S1 { virtual s1func(); }; /* Polymorphic */  
struct __rtti S2 { virtual s2func(); }; /* Polymorphic */  
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism, type information might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations without RTTI, you should declare X with the __rtti modifier. Otherwise, switching the order of the base classes for the class X results in the compile-time error "Can't inherit non-RTTI class from RTTI base 'S1'."

```
struct __rtti S1 { virtual func(); }; /* Polymorphic class */  
struct S2 { }; /* Non-polymorphic class */  
struct __rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with `__rtti`. This makes it safe to mix the order and type of classes.

In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared `__rtti`.

```
struct __rtti S1 { };    // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { };   // The order is not essential
```

Neither the `__rtti` keyword, nor enabling RTTI will make a static class into a polymorphic class.

`__stdcall`, `__stdcall`

Category

Modifiers, Keyword extensions

Syntax

```
__stdcall <function-name>__stdcall
__stdcall <function-name>
```

Description

The `__stdcall` and `__stdcall` keywords force the compiler to generate function calls using the Standard calling convention. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments.

Such functions comply with the standard WIN32 argument-passing convention.

Note: Note: The `__stdcall` modifier is subject to name mangling. See the description of the `-VC` option.

Volatile

Category

Modifiers

Syntax

```
volatile <data definition> ;
```

Description

Use the volatile modifier to indicate that a background routine, an interrupt routine, or an I/O port can change a variable. Declaring an object to be volatile warns the compiler not to make assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile int ticks;
void timer( ) {
    ticks++;
}
void wait (int interval) {
    ticks = 0;
    while (ticks < interval); // Do nothing
}
```


The routines in this example (assuming timer has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument interval. A highly optimizing compiler might not load the value of ticks inside the test of the while loop since the loop doesn't change the value of ticks.

Note: C++ extends volatile to include classes and member functions. If you've declared a volatile object, you can use only its volatile member functions.

Operators

This section contains Operator topics.

In This Section

[__classid](#)

[delete](#)

[If, Else](#)

[new](#)

[Operator](#)

[Sizeof](#)

[Typeid](#)

[__classid](#)

Category

Operators, Keyword extensions

Syntax

```
__classid(classType)
```

Description

The `__classid` operator was added to support the VCL framework. Normally, programmers should not directly use this operator. For more information, see the keyword extensions.

[delete](#)

Category

Operators, C++-Specific Keywords

Syntax

```
void operator delete(void *ptr) throw();
void operator delete(void *ptr, const std::nothrow_t&) throw();
void operator delete[](void *ptr) throw();
void operator delete[](void *ptr, const std::nothrow_t &) throw();
void operator delete(void *ptr, void *) throw(); // Placement form
void operator delete[](void *ptr, void *) throw(); // Placement form
```

Description

The `delete` operator deallocates a memory block allocated by a previous call to `new`. It is similar but superior to the standard library function `free`.

You should use the `delete` operator to remove all memory that has been allocated by the `new` operator. Failure to free memory can result in memory leaks.

The default placement forms of operator `delete` are reserved and cannot be redefined. The default placement `delete` operator performs no action (since no memory was allocated by the default placement `new` operator). If you overload the placement version of operator `new`, it is a good idea (though not strictly required) to provide the overload the placement `delete` operator with the corresponding signature.

If, Else

Category

Operators

Syntax

```
if ( <condition> ) <statement1>;ifelse
if ( <condition> ) <statement1>;
else <statement2>;
```

Description

Use if to implement a conditional statement.

You can declare variables in the condition expression. For example,

```
if (int val = func(arg))
```

is valid syntax. The variable val is in scope for the if statement and extends to an else block when it exists.

The condition statement must convert to a bool type. Otherwise, the condition is ill-formed.

When <condition> evaluates to true, <statement1> executes.

If <condition> is false, <statement2> executes.

The else keyword is optional, but no statements can come between an if statement and an else.

The #if and #else preprocessor statements (directives) look similar to the if and else statements, but have very different effects. They control which source file lines are compiled and which are ignored.

new

Category

Operators, C++-Specific Keywords

Syntax

```
void *operator new(std::size_t size) throw(std::bad_alloc);
void *operator new(std::size_t size, const std::nothrow_t &) throw();
void *operator new[](std::size_t size) throw(std::bad_alloc);
void *operator new[](std::size_t size, const std::nothrow_t &) throw();
void *operator new(std::size_t size, void *ptr) throw(); // Placement form
void *operator new[](std::size_t size, void *ptr) throw(); // Placement form
```

Description

The new operators offer dynamic storage allocation, similar but superior to the standard library function malloc. These allocation functions attempt to allocate size bytes of storage. If successful, new returns a pointer to the allocated memory. If the allocation fails, the new operator will call the new_handler function. The default behavior of new_handler is to throw an exception of type bad_alloc. If you do not want an exception to be thrown, use the nothrow version of operator new. The nothrow versions return a null pointer result on failure, instead of throwing an exception.

The default placement forms of operator new are reserved and cannot be redefined. You can, however, overload the placement form with a different signature (i.e. one having a different number, or different type of arguments). The default placement forms accept a pointer of type void, and perform no action other than to return that pointer, unchanged. This can be useful when you want to allocate an object at a known address. Using the placement form

of new can be tricky, as you must remember to explicitly call the destructor for your object, and then free the pre-allocated memory buffer. Do not call the delete operator on an object allocated with the placement new operator.

A request for non-array allocation uses the appropriate operator new() function. Any request for array allocation will call the appropriate operator new[]() function. Remember to use the array form of operator delete[](), when deallocating an array created with operator new[]().

Note: Arrays of classes require that a default constructor be defined in the class.

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

Operator

Category

Operators, C++-Specific Keywords

Syntax

```
operator <operator symbol>( <parameters> )  
{  
    <statements>;  
}
```

Description

Use the operator keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as *this is implicitly the first argument.

When you overload an operator as a friend, you can specify two arguments.

Sizeof

Category

Operators

Description

The sizeof operator has two distinct uses:

sizeof unary-expression

sizeof (type-name)

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type char (signed or unsigned), sizeof gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals sizeof array/ sizeof array[0] .

If the operand is a parameter declared as array type or function type, sizeof gives the size of the pointer. When applied to structures and unions, sizeof gives the total number of bytes, including any padding.

You cannot use sizeof with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of sizeof is size_t.

You can use `sizeof` in preprocessor directives; this is specific to Borland C++.

In C++, `sizeof(class_type)`, where `class_type` is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Example

```
/* USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>
struct st {
    char *name;
    int age;
    double height;
};
struct st St_Array[] = { /* AN ARRAY OF structs */
    { "Jr.", 4, 34.20 }, /* St_Array[0] */
    { "Suzie", 23, 69.75 }, /* St_Array[1] */
};
int main()
{
    long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };
    printf("\nNumber of elements in LD_Array = %d",
        sizeof(LD_Array) / sizeof(LD_Array[0]));
    /**** THE NUMBER OF ELEMENTS IN THE St_Array. *****/
    printf("\nSt_Array has %d elements",
        sizeof(St_Array)/sizeof(St_Array[0]));
    /**** THE NUMBER OF BYTES IN EACH St_Array ELEMENT. *****/
    printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));
    /**** THE TOTAL NUMBER OF BYTES IN St_Array. *****/
    printf("\nSt_Array=%d", sizeof(St_Array));
    return 0;
}
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 16
St_Array= 32
```

Typeid

Category

Operators, C++-Specific Keywords

Syntax

```
typeid( expression )typeid
typeid( type-name )
```

Description

You can use `typeid` to get runtime identification of types and expressions. A call to `typeid` returns a reference to an object of type `const type_info`. The returned object represents the type of the `typeid` operand.

If the typeid operand is a dereferenced pointer or a reference to a polymorphic type, typeid returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, typeid returns an object that represents the static type.

You can use the typeid operator with fundamental data types as well as user-defined types.

When the typeid operand is a Delphi class object/reference, typeid returns the static rather than runtime type.

If the typeid operand is a dereferenced NULL pointer, the Bad_typeid exception is thrown.

Special Types

This section contains Special Type topics.

In This Section

[Void](#)

Void

Category

Special types

Syntax

```
void identifier
```

Description

void is a special type indicating the absence of any value. Use the void keyword as a function return type if the function does not return a value.

```
void hello(char *name)
{
    printf("Hello, %s.", name);
}
```

Use void as a function heading if the function does not take any parameters.

```
int init(void)
{
    return 1;
}
```

Void Pointers

Generic pointers can also be declared as void, meaning that they can point to any type.

void pointers cannot be dereferenced without explicit casting because the compiler cannot determine the size of the pointer object.

Statements

This section contains Statement topics.

In This Section

[Break](#)
[Case](#)
[Catch](#)
[Continue](#)
[Default](#)
[Do](#)
[__except](#)
[__finally](#)
[For](#)
[Goto](#)
[Return](#)
[Switch](#)
[Throw](#)
[__try](#)
[Try](#)
[While](#)

Break

Category

Statements

Syntax

```
break;
```

Description

Use the break statement within loops to pass control to the first statement following the innermost switch, for, while, or do block.

Case

Category

Statements

Syntax

```
switch ( <switch variable> ){casebreakdefault  
case <constant expression> : <statement>; [break;]  
.  
.  
.  
default : <statement>;  
}
```

Description

Use the case statement in conjunction with switches to determine which statements evaluate.

The list of possible branch points within <statement> is determined by preceding substatements with

```
case <constant expression> : <statement>;
```

where <constant expression> must be an int and must be unique.

The <constant expression> values are searched for a match for the <switch variable>.

If a match is found, execution continues after the matching case statement until a break statement is encountered or the end of the switch statement is reached.

If no match is found, control is passed to the default case.

Note: It is illegal to have duplicate case constants in the same switch statement.

Catch

Category

Statements, C++-Specific Keywords

Syntax

```
catch (exception-declaration) compound-statement
```

Description

The exception handler is indicated by the catch keyword. The handler must be used immediately after the statements marked by the try keyword. The keyword catch can also occur immediately after another catch. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

Continue

Category

Statements

Syntax

```
continue;
```

Description

Use the continue statement within loops to pass control to the end of the innermost enclosing end brace belonging to a looping construct, such as for or while; at which point the loop continuation condition is re-evaluated.

Default

Category

Statements

Syntax

```
switch ( <switch variable> ){casebreakdefault
case <constant expression> : <statement>; [break;]
.
.
.
default : <statement>;
}
```

Description

Use the default statement in switch statement blocks.

- If a case match is not found and the default statement is found within the switch statement, the execution continues at this point.
- If no default is defined in the switch statement, control passes to the next statement that follows the switch statement block.

Do

Category

Statements

Syntax

```
do <statement> while ( <condition> );
```

Description

The do statement executes until the condition becomes false.

<statement> is executed repeatedly as long as the value of <condition> remains true.

Since the condition tests after each the loop executes the <statement>, the loop will execute at least once.

__except

Category

Statements, Keyword extensions

Syntax

```
__except (expression) compound-statement
```

Description

The __except keyword specifies the action that should be taken when the exception specified by expression has been raised.

__finally

Category

Statements, Keyword extensions

Syntax

```
__finally {compound-statement}
```

Description

The `__finally` keyword specifies actions that should be taken regardless of how the flow within the preceding `__try` exits.

The following code fragment shows how to use the `try/__finally` construct:

```
#include <stdio.h>#include <string.h>#include <windows.h>class Exception{public:Exception
(char* s = "Unknown"){what = strdup(s);      }Exception(const Exception& e ){what = strdup
(e.what); } ~Exception()                      {free(what);      } char* msg() const
{return what;                                }private:char* what;};int main(){float e, f, g;try {try    {f = 1.0;
g = 0.0;try        {puts("Another exception:");e = f / g;        }__except
(EXCEPTION_EXECUTE_HANDLER)    {puts("Caught a C-based exception.");throw(Exception
("Hardware error: Divide by 0"));    }    }catch(const Exception& e)    {printf("Caught C
++ Exception: %s :\n", e.msg());    } }__finally {puts("C++ allows __finally too!"); }
return e;}
#include <string.h>
#include <windows.h>
class Exception
{
public:
Exception(char* s = "Unknown"){what = strdup(s);      }
Exception(const Exception& e ){what = strdup(e.what); }
~Exception()                      {free(what);      }
char* msg() const                  {return what;      }
private:
char* what;
};
int main()
{
float e, f, g;
try
{
try
{
f = 1.0;
g = 0.0;
try
{
puts("Another exception:");
e = f / g;
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
puts("Caught a C-based exception.");
throw(Exception("Hardware error: Divide by 0"));
}
}
catch(const Exception& e)
{
printf("Caught C++ Exception: %s :\n", e.msg());
}
}
__finally
{
puts("C++ allows __finally too!");
}
```

```
return e;
}
```

```
#include <iostream>#include <stdexcept>using namespace std;class MyException : public
exception {public:virtual const char *what() const throw() {return("MyException occurred.
");}}; // Give me any integer...void myFunc(int a){ MyException e; // ...but not that one.
if(a == 0) throw(e);}void main(){ int g; // Note __finally must be in its own try
block (with no preceding catch). try { try { g = 0; myFunc(g); }
catch(MyException &e) { cout << "Exception: " << e.what() << endl; } } __finally
{ cout << "Finally block reached." << endl; }
#include <stdexcept>
using namespace std;
class MyException : public exception {
public:
virtual const char *what() const throw() {
return("MyException occurred.");
}
};
// Give me any integer...
void myFunc(int a)
{
MyException e;
// ...but not that one.
if(a == 0)
throw(e);
}
void main()
{
int g;
// Note __finally must be in its own try block (with no preceding catch).
try {
try {
g = 0;
myFunc(g);
}
catch(MyException &e) {
cout << "Exception: " << e.what() << endl;
}
}
__finally {
cout << "Finally block reached." << endl;
}
}
```

Running the above program results in the following:

```
Another exception:Caught a C-based exception.Caught C++ exception[Hardware error: Divide by
0]C++ allows __finally too!Exception: MyException occurred.Finally block reached.
Caught a C-based exception.Caught C++ exception[Hardware error: Divide by 0]C++ allows
__finally too!Exception: MyException occurred.Finally block reached.
```

For

Category

Statements

Syntax

```
for ( [<initialization>] ; [<condition>] ; [<increment>] ) <statement>
```

Description

The for statement implements an iterative loop.

<condition> is checked before the first entry into the block.

<statement> is executed repeatedly UNTIL the value of <condition> is false.

- Before the first iteration of the loop, <initialization> initializes variables for the loop.
- After each iteration of the loop, <increments> increments a loop counter. Consequently, j++ is functionally the same as ++j.

In C++, <initialization> can be an expression or a declaration.

The scope of any identifier declared within the for loop extends to the end of the control statement only.

A variable defined in the for-initialization expression is in scope only within the for-block. See the description of the -Vd option.

All the expressions are optional. If <condition> is left out, it is assumed to be always true.

Goto

Category

Statements

Syntax

```
goto <identifier> ;
```

Description

Use the goto statement to transfer control to the location of a local label specified by <identifier>.

Labels are always terminated by a colon.

Return

Category

Statements

Syntax

```
return [ <expression> ] ;
```

Description

Use the return statement to exit from the current function back to the calling routine, optionally returning a value.

Switch

Category

Statements

Syntax

```
switch ( <switch variable> ) {casebreakdefault
case <constant expression> : <statement>; [break;]
.
.
.
default : <statement>;
}
```

Description

Use the switch statement to pass control to a case that matches the <switch variable>. At which point the statements following the matching case evaluate.

If no case satisfies the condition the default case evaluates.

To avoid evaluating any other cases and relinquish control from the switch, terminate each case with break.

Throw

Category

Statements, C++-Specific Keywords

Syntax

```
throw assignment-expression
```

Description

When an exception occurs, the throw expression initializes a temporary object of the type T (to match the type of argument arg) used in throw(T arg). Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

__try

Category

Statements, Keyword extensions

Syntax

```
__try compound-statement handler-list__try
__try compound-statement termination-statement
```

Description

The __try keyword is supported in both C and C++ programs. You can also use try in C++ programs.

A block of code in which an exception can occur must be prefixed by the keyword __try. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the normal program flow is interrupted. The program begins a search for a handler that matches the exception. If the exception is generated in a C module, it is possible to handle the structured exception in either a C module or a C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution

- Continue the search for some other handler (regenerate the exception)

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

Try

Category

Statements, C++-Specific Keywords

Syntax

```
try compound-statement handler-list
```

Description

The try keyword is supported only in C++ programs. Use `__try` in C programs. C++ also allows `__try`.

A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is transferred to the handler

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

While

Category

Statements

Syntax

```
while ( <condition> ) <statement>
```

Description

Use the while keyword to conditionally iterate a statement.

`<statement>` executes repeatedly until the value of `<condition>` is false.

The test takes place before `<statement>` executes. Thus, if `<condition>` evaluates to false on the first pass, the loop does not execute.

Storage Class Specifiers

This section contains Storage Class Specifier topics.

In This Section

[Auto](#)

[__declspec](#)

[__declspec\(dllexport\)](#)

[__declspec\(dllimport\)](#)

[__declspec\(naked\)](#)

[__declspec\(noreturn\)](#)

[__declspec\(nothrow\)](#)

[__declspec\(novtable\)](#)

[__declspec\(property\)](#)

[__declspec\(selectany\)](#)

[__declspec\(thread\)](#)

[__declspec\(uuid\("ComObjectGUID"\)\)](#)

[Extern](#)

[Mutable](#)

[Register](#)

[Typedef](#)

Auto

Category

Storage class specifiers

Syntax

```
[auto] <data-definition> ;
```

Description

Use the auto modifier to define a local variable as having a local lifetime.

This is the default for local variables and is rarely used.

__declspec

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec(<decl-modifier>)
```

Description

For a list of __declspec keyword arguments used for the VCL framework, see CLXVCL class declarations.

Use the __declspec keyword to indicate the storage class attributes for a variable or function.

The __declspec keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The __declspec keyword and its argument can appear anywhere in the

declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);           // illegal
void __export f(void)           // correct
void __declspec(dllexport) f(void); // correct
__declspec(dllexport) void f(void); // correct
class __declspec(dllexport) ClassName { } // correct
```

In addition to the CLX-related arguments listed above, the supported decl-modifier arguments are:

- dllexport
- dllimport
- naked
- noreturn
- nothrow
- novtable
- property
- selectany
- thread
- uuid

These arguments are equivalent to the following storage class attribute keywords.

Argument	Comparable keyword
dllexport	__export
dllimport	__import
thread	__thread

__declspec(dllexport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllexport ) declarator
```

The `dllexport` storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to export functions, data, and objects from a DLL. This attribute explicitly defines the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as `dllexport` eliminates the need for a module-definition (.DEF) file, at least with respect to the specification of exported functions.

Note: `dllexport` replaces the `__export` keyword.

__declspec(dllimport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllimport ) declarator
```

The `dllimport` storage-class attribute is used for Microsoft C and C++ language compatability. This attribute enables you to import functions, data, and objects to a DLL

Note: Note: `dllimport` replaces the `__import` keyword.

`__declspec(naked)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( naked ) declarator
```

Use of the `naked` argument suppresses the prolog/epilog code. Be aware when using `__declspec(naked)` that it does not set up a normal stack frame. A function with `__declspec(naked)` will not preserve the register values that are normally preserved. It is the programmer's responsibility to conform to whatever conventions the caller of the function expects.

You can use this feature to write your own prolog/epilog code using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

The `naked` attribute is relevant only to the definition of a function and is not a type modifier.

Example

This code defines a function with the `naked` attribute:

```
// Example of the naked attribute
__declspec( naked ) int func( formal_parameters )
{
    // Function body
}
```

`__declspec(noreturn)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( noreturn ) declarator
```

This `__declspec` attribute tells the compiler that a function does not return. As a consequence, the compiler knows that the code following a call to a `__declspec(noreturn)` function is unreachable.

If the compiler finds a function with a control path that does not return a value, it generates a warning. If the control path cannot be reached due to a function that never returns, you can use `__declspec(noreturn)` to prevent this warning or error.

Example

Consider the following code. The else clause does not contain a return statement, so the programmer declares fatal as `__declspec(noreturn)` to avoid an error or warning message.

```
__declspec(noreturn) extern void fatal ()
{
    // Code omitted
}
int foo()
{
    if(...)
        return 1;
    else if(...)
        return 0;
    else
        fatal();
}
```

`__declspec(nothrow)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( nothrow ) declarator
```

This is a `__declspec` extended attribute that can be used in the declaration of functions. This attribute tells the compiler that the declared function and the functions it calls never throw an exception. With the synchronous exception handling model, now the default, the compiler can eliminate the mechanics of tracking the lifetime of certain unwindable objects in such a function, and significantly reduce the code size.

The following three declarations are equivalent:

```
#define WINAPI __declspec(nothrow) __stdcall
```

```
void WINAPI foo1();
```

```
void __declspec(nothrow) __stdcall foo2();
```

```
void __stdcall foo3() throw();
```

Using `void __declspec(nothrow) __stdcall foo2();` has the advantage that you can use an API definition, such as the illustrated by the `#define` statement, to easily specify `nothrow` on a set of functions. The third declaration, `void __stdcall foo3() throw();` is the syntax defined by the C++ standard.

`__declspec(novtable)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( novtable ) declarator
```

This form of `__declspec` can be applied to any class declaration, but should only be applied to pure interface classes, that is classes that will never be instantiated on their own. The `__declspec` stops the compiler from generating code to initialize the vptr in the constructor(s) and destructor of the class. In many cases, this removes the only references to the vtable that are associated with the class and, thus, the linker will remove it. Using this form of `__declspec` can result in a significant reduction in code size

`__declspec(property)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( property( get=get_func_name ) ) declarator  
__declspec( property( put=put_func_name ) ) declarator  
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

This attribute can be applied to non-static “virtual data members” in a class or structure definition. The compiler treats these “virtual data members” as data members by changing their references into function calls.

When the compiler sees a data member declared with this attribute on the right of a member-selection operator (“.” or “->”), it converts the operation to a get or put function, depending on whether such an expression is an l-value or an r-value. In more complicated contexts, such as “+=”, a rewrite is performed by doing both get and put.

This attribute can also be used in the declaration of an empty array in a class or structure definition.

Example

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

The above statement indicates that `x[]` can be used with one or more array indices. In this case:

```
i=p->x[a][b]
```

will be turned into:

```
i=p->GetX(a, b),
```

and

```
p->x[a][b] = i
```

will be turned into

```
p->PutX(a, b, i);
```

`__declspec(selectany)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( selectany ) declarator
```

A global data item can normally be initialized only once in an application or library. This attribute can be used in initializing global data defined by headers, when the same header appears in more than one source file.

Note This attribute can only be applied to the actual initialization of global data items that are externally visible.

Example

This code shows how to use the selectany attribute:

//Correct - x1 is initialized and externally visible

```
__declspec(selectany) int x1=1;
```

//Incorrect - const is by default static in C++, so

//x2 is not visible externally (This is OK in C, since

//const is not by default static in C)

```
const __declspec(selectany) int x2 =2;
```

//Correct - x3 is extern const, so externally visible

```
extern const __declspec(selectany) int x3=3;
```

//Correct - x4 is extern const, so it is externally visible

```
extern const int x4;
```

```
const __declspec(selectany) int x4=4;
```

//Incorrect - __declspec(selectany) is applied to the uninitialized //declaration of x5 extern __declspec(selectany) int x5;

`__declspec(thread)`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( thread ) declarator
```

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process allocates storage for thread-specific data.

The thread extended storage-class modifier is used to declare a thread local variable. The thread attribute must be used with the __declspec keyword.

`__declspec(uuid("ComObjectGUID"))`

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax


```
__declspec( uuid("ComObjectGUID") ) declarator
```

The compiler attaches a GUID to a class or structure declared or defined (full COM object definitions only) with the `uuid` attribute. The `uuid` attribute takes a string as its argument. This string names a GUID in normal registry format with or without the { } delimiters. For example:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
```

```
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

This attribute can be applied in a redeclaration. This allows the system headers to supply the definitions of interfaces such as `IUnknown`, and the redeclaration in some other header (such as `COMDEF.H`) to supply the GUID.

The keyword `__uuidof` can be applied to retrieve the constant GUID attached to a user-defined type.

Extern

Category

Storage class specifiers

Syntax

```
extern <data definition> ;extern  
[extern] <function prototype> ;
```

Description

Use the `extern` modifier to indicate that the actual storage and initial value of a variable, or body of a function, is defined in a separate source code module. Functions declared with `extern` are visible throughout all source files in a program, unless you redefine the function as `static`.

The keyword `extern` is optional for a function prototype.

Use `extern "C"` to prevent function names from being mangled in C++ programs.

Mutable

Category

C++-Specific Keywords, Storage class specifiers

Syntax

```
mutable <variable name>;
```

Description

Use the `mutable` specifier to make a variable modifiable even though it is in a `const`-qualified expression.

Using the mutable Keyword

Only class data members can be declared `mutable`. The `mutable` keyword cannot be used on `static` or `const` names. The purpose of `mutable` is to specify which data members can be modified by `const` member functions. Normally, a `const` member function cannot modify data members.

Register

Category

Storage class specifiers

Syntax

```
register <data definition> ;
```

Description

Use the register storage class specifier to store the variable being declared in a CPU register (if possible), to optimize access and reduce code.

Note: The compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

Typedef

Category

Storage class specifiers

Syntax

```
typedef <type definition> <identifier> ;
```

Description

Use the typedef keyword to assign the symbol name <identifier> to the data type definition <type definition>.

Type Specifiers

This section contains Type Specifier topics.

In This Section

[Char](#)
[Class](#)
[Double](#)
[Enum](#)
[Float](#)
[Int](#)
[Long](#)
[Short](#)
[Signed](#)
[Struct](#)
[Union](#)
[Unsigned](#)
[Wchar_t](#)

Char

Category

Type specifiers

Syntax

```
[signed|unsigned] char <variable_name>
```

Description

Use the type specifier char to define a character data type. Variables of type char are 1 byte in length.

A char can be signed, unsigned, or unspecified. By default, signed char is assumed.

Objects declared as characters (char) are large enough to store any member of the basic ASCII character set.

Class

Category

C++-Specific Keywords, Type specifiers

Syntax

```
<classkey> <classname> <baselist> { <member list> }
```

- <classkey> is either a class, struct, or union.
- <classname> can be any name unique within its scope.
- <baselist> lists the base class(es) that this class derives from. <baselist> is optional
- <member list> declares the class's data members and member functions.

Description

Use the `class` keyword to define a C++ class.

Within a class:

- the data are called data members
- the functions are called member functions

Double

Category

Type specifiers

Syntax

```
[long] double <identifier>
```

Description

Use the `double` type specifier to define an identifier to be a floating-point data type. The optional modifier `long` extends the accuracy of the floating-point value.

If you use the `double` keyword, the floating-point math package will automatically be linked into your program.

Enum

Category

Type specifiers

Syntax

```
enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];
```

- `<type_tag>` is an optional type tag that names the set.
- `<constant_name>` is the name of a constant that can optionally be assigned the value of `<value>`. These are also called enumeration constants.
- `<value>` must be an integer. If `<value>` is missing, it is assumed to be: `<prev> + 1` where `<prev>` is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0.
- `<var_list>` is an optional variable list that assigns variables to the enum type.

Description

Use the `enum` keyword to define a set of constants of type `int`, called an enumeration data type.

An enumeration data type provides mnemonic identifiers for a set of integer values. Use the `-b` flag to toggle the Treat Enums As Ints option. Enums are always interpreted as ints if the range of values permits this, but if they are not ints the value gets promoted to an int in expressions. Depending on the values of the enumerators, identifiers in an enumerator list are implicitly of type signed char, unsigned char, short, unsigned short, int, or unsigned int.

In C, an enumerated variable can be assigned any value of type `int`--no type checking beyond that is enforced. In C++, an enumerated variable can be assigned only one of its enumerators.

In C++, you can omit the `enum` keyword if `<tag_type>` is not the name of anything else in the same scope. You can also omit `<tag_type>` if no further variables of this enum type are required.

In the absence of a <value> the first enumerator is assigned the value of zero. Any subsequent names without initializers will then increase by one. <value> can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers.

In C++, enumerators declared within a class are in the scope of that class.

Float

Category

Type specifiers

Syntax

```
float <identifier>
```

Description

Use the float type specifier to define an identifier to be a floating-point data type.

Type	Length	Range
float	32 bits	$3.4 * (10^{**-38})$ to $3.4 * (10^{**+38})$

The floating-point math package will be automatically linked into your program if you use floating-point values or operators.

Int

Category

Type specifiers

Syntax

```
[signed|unsigned] int <identifier> ;
```

Description

Use the int type specifier to define an integer data type.

Variables of type int can be signed (default) or unsigned.

Long

Category

Type specifiers

Syntax

```
long  
[int] <identifier> ;longdouble [long] double <identifier> ;
```

Description

When used to modify a double, it defines a floating-point data type with 80 bits of precision instead of 64.

The floating-point math package will be automatically linked with your program if you use floating-point values or operators.

Short

Category

Type specifiers

Syntax

```
short int <variable> ;
```

Description

Use the short type modifier when you want a variable smaller than an int. This modifier can be applied to the base type int.

When the base type is omitted from a declaration, int is assumed.

Signed

Category

Type specifiers

Syntax

```
signed <type> <variable> ;
```

Description

Use the signed type modifier when the variable value can be either positive or negative. The signed modifier can be applied to base types int, char, long, short, and __int64.

When the base type is omitted from a declaration, int is assumed.

Struct

Category

Type specifiers

Syntax

```
struct [<struct type name>] {  
    [<type> <variable-name[, variable-name, ...]>] ;  
    .  
    .  
    .  
} [<structure variables>] ;
```

Description

Use a struct to group variables into a single record.

<struct type name> An optional tag name that refers to the structure type.

<structure variables> The data definitions, also optional.

Though both <struct type name> and <structure variables> are optional, one of the two must appear.

You define elements in the record by naming a <type>, followed by one or more <variable-name> (separated by commas).

Separate different variable types by a semicolon.

Use the . operator, or the -> operator to access elements in a structure.

To declare additional variables of the same type, use the keyword struct followed by the <struct type name>, followed by the variable names. In C++ the keyword struct can be omitted.

Note: The compiler allows the use of anonymous struct embedded within another structure.

Union

Category

Type specifiers

Syntax

```
union [<union type name>] {  
    <type> <variable names> ;  
    ...  
} [<union variables>] ;
```

Description

Use unions to define variables that share storage space.

The compiler allocates enough storage in a_union to accommodate the largest element in the union.

Unlike a struct, the members of a union occupy the same location in memory. Writing into one overwrites all others.

Use the record selector (.) to access elements of a union .

Unsigned

Category

Type specifiers

Syntax

```
unsigned <type> <variable> ;
```

Description

Use the unsigned type modifier when variable values will always be positive. The unsigned modifier can be applied to base types int, char, long, short, and __int64.

When the base type is omitted from a declaration, int is assumed.

Wchar_t

Category

Syntax

```
wchar_t <identifier>;
```

Description

In C++ programs, `wchar_t` is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In Borland C++, A `wchar_t` type is the same size, signedness, and alignment requirement as an unsigned short type.

Together Reference

This section contains links to the reference material for UML modeling with Together.

In This Section

[Together Glossary](#)

Main terms of Together products.

[GUI Components for Modeling](#)

[Together Wizards](#)

[Together Keyboard Shortcuts](#)

Describes Together keyboard shortcuts.

[Together Configuration Options](#)

[UML 1.5 Reference](#)

[UML 2.0 Reference](#)

[Together Refactoring Operations](#)

Describes Together refactoring operations.

[Project Types and Formats with Support for Modeling](#)

About project formats supported by Together.

Together Glossary

This topic contains a dictionary of specific terms used in Together user interface and documentation. This dictionary is sorted alphabetically.

Term	Description
Cardinality	The number of elements in a set. See also multiplicity.
Classifier	In general, a classifier is a classification of instances — it describes a set of instances that have features in common. In Together, classifiers are the basic nodes of Class diagrams: class, interface, structure, delegate, enum, module. Some of them can have so called inner classifiers, i. e. they can include some other classifiers (see Inner Classifiers).
Compartment	Some of Together model elements (basically, classes) are represented by rectangles with several compartments inside. You can change appearance of the compartments (see Diagram Appearance options).
Diagram	A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). The set of available diagram for a project depend on the project type.
Invocation specification	See Invocation Specification.
Model element	Model element is any component of your model that you can put on a diagram. Model elements include nodes and links between them.
Multiplicity	A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for association ends, parts within composites, repetitions, and other purposes. A multiplicity is a subset of the non-negative integers. See also cardinality.
N-ary association	An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes.
Shortcut	A shortcut is a representation of an existing node element placed on the same or a different diagram.
View filter	A view filter is mechanism to show or hide a specific kind of model elements. When dealing with large projects, the amount of information shown on a diagram can become overwhelming. In Together, you can selectively show or hide information. See Using view filters.

GUI Components for Modeling

This section describes GUI components of the Developer Studio 2006 interface you use for UML modeling.

Menus

Item	Description
File menu	You can use the File menu to export diagrams to image files, and print diagrams.
Edit menu	Use the Edit menu to cut, copy, paste, and delete diagrams and diagram elements, select all items on a diagram, and undo/redo actions.
View menu	The View menu contains the command for opening the Model View.
Project menu	Use the Project menu to enable or disable Together support for specific projects in the project group currently open.
Refactor menu	The Refactor menu contains refactoring commands for the implementation projects.
Tools menu	Use the Tools menu to generate documentation, open the pattern registry and pattern organizer, run audits and metrics (for the implementation projects), and set Together-specific options.
Diagram context menu	Use the Diagram context menu to add new elements, manage the layout, zoom in and out, show or hide diagram elements, synchronize your diagram with the Model view, and edit hyperlinks.
Model View context menu	Context menus of the various elements of the Model View are context-sensitive. The list of elements that can be added to a diagram from the Model View depends on the element and diagram type.
Element context menus	You can add or delete members (or delete the element itself), cut/copy/paste, view source code, show element information and more. Explore the context menus of the different elements as you encounter them to see what is available for each one.

Tool Palette

View ► Tool Palette

Together extends the **Tool Palette** of Developer Studio 2006 by adding model elements to it.

The Developer Studio 2006 **Tool Palette** displays special tabs for the supported UML diagrams. When a diagram is opened in the **Diagram View**, the appropriate tab appears in the **Tool Palette**.

In the **Tool Palette** you see model elements (nodes, links) that can be placed on the current diagram. However, you can choose to show the tabs for all diagram types. Use the **Tool Palette** buttons to create the diagram contents.

Note: The set of available model elements depends on the type of a diagram that is currently selected in the **Diagram View**. For descriptions of available elements, refer to Together Reference.

Tip: You can control the diagram elements that appear in the **Tool Palette** and create your own **Tool Palette** tabs with specified items. You can cut, copy, paste, delete, rename, move up, and move down **Tool Palette** items to customize your **Tool Palette** view. You can also display **Tool Palette** items alphabetically or in a list. Use the **Tool Palette** context menu to accomplish such tasks. For further information, please refer to the Developer Studio 2006 documentation.

Object Inspector

View ► Object Inspector

When Together support is activated, the **Object Inspector** shows the properties of an element that is selected in the Model or Diagram Views. To view the **Object Inspector**, choose **Object Inspector** from the **View** menu, press F4, or press ALT+ENTER. The content of the **Object Inspector** depends on the element type.

You can use the **Object Inspector** to edit diagram or element properties.

There are several categories of properties:

Item	Description
Description	Text editor field where you can optionally provide textual description of an element.
Design	These properties are used to define the appearance of an element.
General	UML and source code element properties.
User properties	This node appears when there are user properties defined. In addition to the standard properties, you can define an unlimited number of user-defined properties.

Using the **Object Inspector**, you can view and edit the properties of an element. Clicking on an editable field reveals which type of internal editor is available: text area, combobox with a list of values, or a dialog box. The read-only fields are displayed gray. As you click on the element properties, their respective descriptions display at the bottom of the **Object Inspector**.

Diagram View

Context menu (in the Model View) ▶ Open Diagram

The **Diagram View** displays model diagrams. Each diagram is presented in its own tab.

To open the **Diagram View**, choose a diagram, namespace or a package in the **Model View**, right-click it and choose **Open Diagram** on the context menu.

Most manipulations with diagram elements and links involve drag-and-drop operations or executing right-click (or context) menu commands on the selected elements.

Some of the actions provided by the context menus are:

- Add or delete diagram elements and links
- Add or delete members in the elements
- Create elements by pattern
- Cut, copy, and paste the selected items
- Navigate to the source code
- Hyperlink diagrams
- Zoom in and out

Item	Description
Working area	The main part of the Diagram View shows the current diagram.
Context menu	The context menus of the Diagram View are context-sensitive. Right-clicking model elements, including class members, provides access to element-specific operations on the respective context menu. Right-clicking the diagram background opens the context menu for the diagram.
Overview button	Opens the Overview pane (see below).

Overview pane

The overview feature of the **Diagram View** provides a thumbnail view of the current diagram. The Overview button is located in the bottom right corner of every diagram.

OCL Editor

The OCL Editor is used to enter and edit OCL expressions. Any changes to the names of your model components (classes, operations, attributes, and so on) used in these expressions are automatically updated by Together. This guarantees that your OCL constraints always stay up-to-date.

Model View

View ▶ Model View

The **Model View** provides the logical representation of the model of your project: namespaces (packages) and diagram nodes. Using this view, you can add new elements to the model; cut, copy, paste and delete elements, and more. Context menu commands of the **Model View** are specific to each node. Explore these commands as you encounter them.

In the **Model View**, only the nodes and their respective subnodes shown in the **Diagram View** are listed under the corresponding diagram node. For example, if you have a namespace (package) containing a class, both the namespace (package) and class are shown under the diagram node in the **Model View**. However, any members of the class are not shown under the diagram node as they are displayed under the namespace (package) node only.

Although the **Model View** opens initially as a free-floating window, it is a dockable window. The docking areas are any of the four borders of the Developer Studio 2006 window. You can position the **Model View** window according to your preferences.

The following options are applicable to the **Model View**:

- In the **Show diagram nodes expandable** field (**Options ▶ Together ▶ Model View**) choose *True* to show, or *False* to hide expandable diagram nodes. By default, the **Model View** displays expandable diagram nodes with the elements contained therein. You can hide expandable diagram nodes to further simplify the visual presentation of the project.
- In the **Show links** field (**Options ▶ Together ▶ Model View**) choose *True* to show, or *False* to hide expandable diagram nodes. By default, the Model View does not display links between nodes. You can opt to show the links to make the visual presentation of the project more detailed.
- In the **Sorting type** field (**Options ▶ Together ▶ Model View**) choose *Metaclass*, *Alphabetical*, or *None* to sort elements in the **Model View**. By default, diagram nodes are sorted by metaclass. You can sort elements in the **Model View** by metaclass, alphabetically, or none.
- In the **View type** field (**Options ▶ Together ▶ Model View**) choose Diagram-centric or Model-centric from the list box. For the sake of better presentation you can opt to show your model in diagram-centric or in model-centric modes. The Diagram-centric mode assumes that the design elements are shown under their respective diagrams; the namespaces only contain classes and interfaces (and the source-code elements for implementation projects). The Model-centric mode assumes that all elements are shown under the namespaces.

Item	Description
Root project node	The topmost item of a project structure.
Nodes	Namespaces (packages), diagrams, then model elements of the current model.
Refresh Model View button	Updates the model structure in the Model View to show possible changes in source code.
Regenerate ECO source code button	
Update ECO source code button	

Reload command

This command is available for implementation projects only.

You can use the **Reload** command (available at the root project node) to refresh the Together model from the source code. This command provides a total refresh for the elements and removes invalid code elements from the model. Using this command has the same effect as reopening the project group, but avoids the overhead of reinitializing Developer Studio 2006.

Pattern GUI Components

This section describes GUI components of the Developer Studio 2006 interface you use for Together Pattern features.

Pattern Organizer

Tools ► Pattern Organizer

The Pattern Organizer window enables you to logically organize the patterns found in the Pattern Wizard using virtual trees, folders and shortcuts. You can also view and edit pattern properties.

Virtual pattern trees

This section displays the logical hierarchy of patterns. Context menus are provided for the root folder node, subfolders, and the pattern elements. The root folder context menu items are as follows:

Menu item	Description
New Pattern Tree	Use this command to create a new Pattern Tree node.
Sort Folder	Sorts nodes in ascending alphabetical order.

The subfolders beneath the root folder contain the following context menu items:

Menu item	Description
New Folder	Use this command to create a new subfolder under the selected folder.
New Shortcut	Opens the Pattern Registry allowing you to create a new shortcut to a pattern. The shortcut is placed in the selected folder.
Cut	Cuts the selected node to the clipboard.
Copy	Copies the selected node to the clipboard.
Paste	Pastes the clipboard contents to the selected node.
Delete	Removes the selected node.
Sort Folder	Sorts nodes in ascending alphabetical order.

The context menu for pattern elements contains the following menu items:

Assign Pattern	Opens the Pattern Registry allowing you to assign a pattern to the selected pattern element.
Cut	Cuts the selected node to the clipboard.
Copy	Copies the selected node to the clipboard.
Paste	Pastes the clipboard contents to the selected node.
Delete	Removes the selected node.

Properties

This section displays properties of the selected pattern or folder. You can edit the Name and Visible fields.

Name	The name displayed in the Virtual pattern tree. This field is editable.
Valid	This field applies only to patterns. If you have registered the pattern using the Pattern Registry , then the status is reported as valid. Otherwise, the pattern status is Invalid, and it will not display in the Pattern Wizard . Folders are always considered valid and shown in the Pattern Wizard dialog (unless hidden using the Visible property).
Visible	Use the combobox to specify whether the pattern or folder is visible or hidden in the Pattern Wizard dialog.

Pattern description

A read-only section that displays comments for the selected pattern.

Edit Shared Patterns Root

Click this button to open the list of shared patterns roots. Use **Add** and **Remove** buttons to make up the desired list, and click **OK** when ready.

Pattern Registry

Pattern Organizer context menu ► New Shortcut (or: Assign Pattern)

The **Pattern Registry** defines the virtual hierarchy of patterns. When you create a folder or a shortcut in the Pattern Organizer and save the changes, a new entry is added to the registry of shortcuts. All operations with the contents of the Pattern Registry are performed in the **Pattern Organizer**, and synchronized with the registry.

The window opens when using **New Shortcut** and **Assign Pattern** commands on the **Pattern Organizer** context menu.

Filters

The filters determine which patterns to display in the Patterns table. The Filters section provides the following set of filtering options:

Option	Description
Category	The available options are: Class Link Member Other All
Register	This option filters patterns by status of registration: All - all existing patterns Already - the patterns assigned to shortcuts None - the patterns not assigned to shortcuts
Containers	This option filters patterns by the container metaclass.
Diagram type	This option filters patterns that pertain to the selected diagram type.
Language	This option is available for implementation projects only, and enables you to filter out language-specific patterns.

Main sections

Option	Description
Patterns	Displays a table with the names and types of available patterns.
Pattern Properties	Displays the properties of the pattern selected in the Patterns table.
Pattern Description	A read-only section that displays comments for the pattern selected in the Patterns table.

Buttons

Button	Description
Synchronize	Click this button to search for patterns throughout your storage and update the Pattern Registry .
OK	Click this button to save the filtering settings and close the window.
Cancel	Click this button to discard the filtering settings and close the window.
Help	Displays this page.

Quality Assurance GUI Components

This section describes GUI components of the Developer Studio 2006 interface you use for Together Quality Assurance features.

Audit Results Pane

QA Audits dialog window ► Start button

Use this pane to view and export audit results.

Audit results are displayed as a table in the Audits View.

Each time that you generate audits for the same project, the audit results display in a tabbed-page format with the most recent results having focus in the window.

Although the audit results open initially as a free-floating window, it is a dockable window. The docking areas are any of the four borders of the Developer Studio 2006 window. You can position the audit results window according to your preferences.

Tip: Press **F1** with the Audits View having the focus to display this page.

Item	Description
Toolbar buttons	
Save Audit results	Saves audit results.
Print Audit results	Prints audit results.
Refresh	Recalculates the results that are currently displayed.
Restart	Opens the Audits dialog window, define new settings and start new audits analysis.
Context menu commands	
Group By	Groups items according to the selected column.
Show Description	Displays a window with the full name and description of the selected audit.
Open	Opens the selected element in the source code editor highlighting the relevant code.
Copy	You can copy one row or multiple rows in the Audit results.
	Use CTRL+CLICK to select multiple rows to copy.
Close	Closes the current tab.
Close All	Closes all tabs and the results window.
Close All But This	Closes all tabs except for the tab currently in focus.

Note that only audit violations are shown in the results table. For this reason, the results do not necessarily display all of the audits that you ran, or all the packages or classes that you processed. The table contains the following columns:

Audit violations

Violation table column	Description
Abbreviation	The abbreviation for the audit name. The full name is displayed in the description (choose Show Description on the context menu of the violation).
Description	Describes why the audit flagged the item.
Severity	Indicates how serious, in general, violations of the audit are considered to be. This will help you sort the results and assess which violations are critical and which are not.
Resource	The source code item that was flagged by the audit.
File	The file that contains the problem code.
Line	The line number in the file where the problem code is located.

Metric Results Pane

[QA Metrics dialog window](#) ► [Start button](#)

Use this pane to view and export metric results.

The metrics results report is displayed as a table in the **Metrics results pane**. The rows display the elements that were analyzed, and the columns display the corresponding values of selected metrics. Context menus of the rows and columns enable you to navigate to the source code, view descriptions of the metrics, and produce graphical output.

Tip: Press **F1** with the **Metrics results pane** having the focus to display this page.

Item	Description
Toolbar buttons	
Save	Saves metric results.
Refresh	Recalculates the results that are currently displayed.
Restart	Opens the Metrics dialog window, define new settings and start new metrics analysis.
Context menu commands	
Show Description	Displays a window with the full name and description of the selected metric.
Chart	Builds a metric chart.

Together Wizards

This section describes Wizards used for UML modeling.

New Together Project Wizards

This section describes Wizards used to create new Together modeling projects.

UML 1.5 Together Design Project Wizard

File ► **New** ► **Other** ► **Design Projects** ► **UML 1.5 Design Project**

This project is provided by Together for creation of a UML 1.5 design project.

To start this wizard, choose **File** ► **New** ► **Other** on the main menu. The **New Items** dialog box opens. Choose the **Design Projects** ► **UML 1.5 Design Project** category. Click **OK**.

The **New Application** dialog box opens. Specify the name and location of your model project. Click **OK**.

The new UML 1.5 design project is created. Use the **Model View** to see its structure.

UML 2.0 Together Design Project Wizard

File ► **New** ► **Other** ► **Design Projects** ► **UML 2.0 Design Project**

This project is provided by Together for creation of a UML 2.0 design project.

To start this wizard, choose **File** ► **New** ► **Other** on the main menu. The **New Items** dialog box opens. Choose the **Design Projects** ► **UML 2.0 Design Project** category. Click **OK**.

The **New Application** dialog box opens. Specify the name and location of your model project. Click **OK**.

The new UML 2.0 design project is created. Use the **Model View** to see its structure.

Convert MDL Wizard

Use this wizard to create a design project around an existing IBM Rational Rose (MDL) model. The wizard is invoked by the [Design Projects](#) ► [Convert from MDL](#) template of the **New Project** dialog box.

Paths section

Button	Description
Add	Adds one model file to the Paths section. Press this button to open Select Model File dialog box, navigate to the desired model file and click Open.
Add Folder	Adds all model files in the selected folder. Press this button to open Browse for Folder dialog box, navigate to the desired folder that contains the model files, and click OK.
Remove	Press this button to delete the selected entry from the Paths section.
Remove all	Press this button to delete all model files from the Paths section.

Options section

Option	Description
Scale factor	Specify the element dimensions coefficient. By default, the scale factor is 0.3.
Convert Rose default colors	If this option is checked, the default Rational Rose will be replaced with the default Together colors.
Preserve diagram nodes bounds	if this option is checked, user-defined bounds are preserved in the resulting diagrams. Otherwise the default values are applied.
Convert Rose actors	This options enables you to choose mapping for the Rose classes with actor-like stereotypes (Actor, Business Actor, Business Worker, Physical Worker.) If the option is checked, the Rose actors are mapped to Together actors. If the option is not checked, the Rose actors are mapped to the classes with the Actor stereotype.

Supported Delphi Project Wizards

Together supports all Delphi project types available in Developer Studio 2006.

Such projects can be supplied with a UML 1.5 model.

Please consult the general documentation for Developer Studio 2006 for further information about creating Delphi projects.

Supported C# Project Wizards

Together supports all C# project types available in Developer Studio 2006.

Such projects can be supplied with a UML 1.5 model.

Please consult the general documentation for Developer Studio 2006 for further information about creating C# projects.

Pattern Wizard

The **Pattern Wizard** enables you to explicitly apply a pattern. You can open the **Pattern Wizard** by:

- Using the **Node by Pattern** or **Link by Pattern** button in the **Tool Palette**
- Using the **Create by Pattern** command from the **Diagram View** or class context menus

The Node by Pattern element opens the **Pattern Wizard**. You can also open the **Pattern Wizard** using the Create by Pattern command on the diagram context menu.

Item	Description				
Pattern tree:	This field determines the content displayed in the Patterns pane. Use the drop-down arrow to select a pattern tree. Pattern trees are defined in the Pattern Organizer.				
Panes	The following Selector/Editor panes occupy the top portion of the dialog: <table><tr><td>Patterns:</td><td>The Patterns pane presents a tree view of the available patterns. Use the Pattern tree field to determine the available content shown in the Patterns pane.</td></tr><tr><td>Pattern properties:</td><td>Edit the generated class names for pattern elements, or use the information button to open the Select Element dialog for choosing elements.</td></tr></table>	Patterns:	The Patterns pane presents a tree view of the available patterns. Use the Pattern tree field to determine the available content shown in the Patterns pane.	Pattern properties:	Edit the generated class names for pattern elements, or use the information button to open the Select Element dialog for choosing elements.
Patterns:	The Patterns pane presents a tree view of the available patterns. Use the Pattern tree field to determine the available content shown in the Patterns pane.				
Pattern properties:	Edit the generated class names for pattern elements, or use the information button to open the Select Element dialog for choosing elements.				
Description	The Description pane lies at the bottom of the Pattern Wizard and displays context-sensitive help text. Help descriptions for the more complex patterns appear directly in the Pattern Wizard rather than in Together online help. To view a description of a pattern, click on the individual pattern in the Patterns pane.				
Error	If an error occurs while applying a pattern, the Pattern Wizard remains open, and the error text displays here.				
Buttons					
OK	Applies the specified pattern.				
Cancel	Closes the Pattern Wizard without applying a pattern.				

Create Pattern Wizard

The **Create Pattern Wizard** enables you to save an existing fragment of your model as a pattern for future use. You can open the **Create Pattern Wizard** by selecting one or more nodes and links on a Class diagram and choosing the **Save as Pattern** command on a context menu.

Item	Description
File	This field specifies the target XML file name.
Name	This field specifies the name of the new pattern.
Create Pattern Object	Check this check box to use your pattern as a First Class Citizen. This means that an oval pattern element will display on your diagrams when applying the pattern.

Together Keyboard Shortcuts

Together enables you to perform many diagram actions without using the mouse. You can navigate between diagrams, create diagram elements, and more, using the keyboard only.

Navigational shortcut keys

Keyboard shortcuts for navigation and browsing:

Action	Shortcut	Notes
Navigate between open diagrams in the Diagram View	CTRL+Tab	The title of the diagram that has focus is in bold text.
Expand node in Model View	Right arrow	
Collapse node in Model View	Left arrow	
Open the Object Inspector	F4, or Alt + Enter	
Close current diagram	CTRL+F4	
Toggle between a selected container node and its members	PgDown/PgUp	
Navigate between nodes or node members	Arrow keys, Shift + arrow keys	

Shortcut keys for editing

Keyboard shortcuts for editing:

Action	Shortcut
Cut, Copy, or Paste model elements or members.	CTRL+X, CTRL+C, CTRL+V
Activate the in-place editor for a diagram element to edit, rename a member.	F2
Undo	CTRL+Z
Redo	CTRL+Y, CTRL+SHIFT+Z
Select all elements on the diagram	CTRL+A
Close the Overview window	ESC
Add a new namespace (package) to a diagram	CTRL+E
Add a new class to a diagram	CTRL+L
Add new method (operation) to a class or interface	CTRL+M
Add a new field (attribute) to a class	CTRL+W
Add a new interface to diagram	CTRL+SHIFT+L
Open the Add Shortcuts dialog box	CTRL+SHIFT+M
Add a new diagram from the Model View	CTRL+SHIFT+D

Zoom shortcut keys

Keyboard shortcuts for zooming the diagram image:

Action	Shortcut	Notes
Zoom in	+	Use the numeric keypad
Zoom out	-	Use the numeric keypad
Fit the entire diagram in the Diagram View	*	Use the numeric keypad
Display the actual size	/	Use the numeric keypad

Other shortcut keys

Other keyboard shortcuts:

Action	Shortcut
Open the Print Diagram dialog box	CTRL+P
Diagram update	F6

Together Configuration Options

This section describes UML modeling options.

Configuration Levels

The configuration options apply to the four hierarchical levels. Each level contains a number of categories, or groups, of options.

The configuration options can be specified at the following levels:

- **Default:** Options at this level apply to all the current projects and project groups, as well as newly created ones. Diagram options apply to newly created diagrams within these projects and project groups. All options are available at this level.
- **Project group:** Options at this level apply to the current project group. Diagram options apply to newly created diagrams within this project group. All options are available at this level.
- **Project:** Options at this level apply to the current project. Diagram options apply to newly created diagrams within this project. All options except the Model View category are available at this level.
- **Diagram:** Options at this level apply to the current diagram. The Diagram options category is only available at this level.

Option Value Editors

To edit an option, click the value field to invoke the appropriate value editor. There are several types of value editors:

- **In-line text editor.** To edit a value in a text field, type the new value. Changes are applied when you press Enter or move the focus to another field.
- **Combo box or list box.** Clicking a box field reveals the list of possible values. Select the required value from the list.
- **Dialog box.** Clicking a dialog box field reveals the button that opens the dialog box. Specify the required values and click OK to apply changes.

Together Option Categories

This section describes modeling option categories.

Together General Options

The General options allow you to customize certain behaviors in the user interface that do not pertain to any other specific category of options such as Diagram or View Filters. The table below lists the General options, descriptions, and default values.

General Group

Option	Description and default value
Delete Confirmation	This option defines whether confirmation is requested before deleting a namespace, classifier, or diagram. The default value is True.

Together Support Group

Option	Description and default value
Automatically enable Together support for new and opened projects	This option defines whether Together support is automatically enabled for opened and new projects added to an existing project group. A project is considered new when it is created within an existing project group using File New Project. Note that Together support, enabled using the Model Support dialog, overrides this setting. The default value is True.

Together Diagram Appearance Options

The Diagram options enable you to control a number of default behaviors and appearances of diagrams. The table below lists the Appearance options, descriptions, and default values.

General group

Option	Description and default value						
Custom diagram background color	This parameter controls the background color of diagrams, if the Use default background color option is set to false. The default value is WhiteSmoke.						
Diagram detail level	This option defines how much information will be displayed for an element in diagrams. The default value is Design. <table> <tr> <td>Design</td><td>names and types (visibility signs are shown)</td></tr> <tr> <td>Analysis</td><td>names only (no visibility signs)</td></tr> <tr> <td>Implementation</td><td>Names and types, parameters for the methods, and initial values of attributes (visibility signs are shown)</td></tr> </table>	Design	names and types (visibility signs are shown)	Analysis	names only (no visibility signs)	Implementation	Names and types, parameters for the methods, and initial values of attributes (visibility signs are shown)
Design	names and types (visibility signs are shown)						
Analysis	names only (no visibility signs)						
Implementation	Names and types, parameters for the methods, and initial values of attributes (visibility signs are shown)						
Font in diagrams	This option defines the font and font size that is used in printed diagrams. The default value is Arial, 9.75pt.						
Wrap text in nodes	This option controls whether the text displayed in a node automatically continues on the next line when it reaches the right border of the node. If set to False, the rest of the text is not displayed. The default value is True.						
Maximum width of text labels (pixels)	This setting specifies a width limit for all text labels outside nodes (for example, link labels). The text automatically continues on the next line when it reaches this limit. If set to 0, the text is always displayed in a single line. The default value is 200.						
Member format	This option controls the format of members in class diagrams. The default value is UML. <table> <tr> <td>UML</td><td>Methods (functions) are displayed as <code><name> (parameters) : <type></code>. Fields are displayed as <code><name> : type</code></td></tr> <tr> <td>Language</td><td>Methods in C# are displayed as <code><type> <name></code>. Fields in C# are displayed as <code><type> <name></code>.</td></tr> </table>	UML	Methods (functions) are displayed as <code><name> (parameters) : <type></code> . Fields are displayed as <code><name> : type</code>	Language	Methods in C# are displayed as <code><type> <name></code> . Fields in C# are displayed as <code><type> <name></code> .		
UML	Methods (functions) are displayed as <code><name> (parameters) : <type></code> . Fields are displayed as <code><name> : type</code>						
Language	Methods in C# are displayed as <code><type> <name></code> . Fields in C# are displayed as <code><type> <name></code> .						
Show page borders	This option controls whether to show gray borders that represent page margins in the Diagram View and Overview. The default value is False.						
Use default background color	This parameter controls whether the system background color is used in diagrams. If this parameter is true, then background color is defined according to the current Windows color scheme. If this parameter is false, the background color is defined by the Custom diagram background color parameter (see above). The default value is True.						

Grid group

Option	Description and default value
Grid color	This parameter defines the color of the diagram grid. The default value is LightGray.
Grid height (in pixels)	This option enables you to specify the exact height of grid squares in pixels. The default value is 10.
Grid style	This parameter controls whether the grid is displayed as dotted or solid lines. The default value is Lines.
Grid width (in pixels)	This option enables you to specify the exact width of grid squares in pixels. The default value is 10.
Show grid	If this option is true, a design grid is visible in the background behind diagrams. The default value is True.
Snap to grid	If this option is true, diagram elements "snap" to the nearest coordinate of the diagram background design grid. The snap function works whether the grid is visible or not. The default value is True.

Nodes group

Option	Description and default value
3D look	If this option is true, a shadow appears under each diagram element to create a three-dimensional effect. The default value is True.
Show compartments as line	If this option is true, a control bar displays over the compartments of selected diagram elements (fields, methods, classes, and properties). The compartments are represented as expandable nodes that can be opened or closed by a mouse click. The default value is True.
Show imported classes with fully qualified names	This parameter controls whether imported class names are shown in the fully qualified or short form. The default value is True.
Show referenced class names	With this parameter, you can optionally show or hide the name of the base class or interface in the top-right corner of a classifier in the Diagram View. You can hide these references to simplify the visual presentation of the project. The default value is True.
Show referenced classes with fully qualified names	This parameter controls whether referenced class names are shown in the fully qualified or short form. The default value is True.
Sort according to positions in source code	Choose this option to sort fields, methods, subclasses and properties according to their positions in the source code. This option prevails over the others: if it is set to True, the options Sort elements alphabetically and Sort elements by visibility are ignored.

Sort elements alphabetically	<p>This option controls the order of members displayed within elements on diagrams. If this option is true, fields, methods, subclasses, and properties are sorted alphabetically within compartments.</p> <p>The default value is True.</p>
Sort elements by visibility	<p>This option controls the order of members displayed within elements on diagrams. If this option is true, fields, methods, subclasses, and properties are sorted by visibility within compartments.</p> <p>The default value is True.</p>

UML In Color group

Option	Description and default value
Description stereotype	<p>This parameter controls the color of classifiers with the stereotype "description."</p> <p>The default value is Light blue.</p>
Mi-detail stereotype	<p>This parameter controls the color of classifiers with the stereotype "Mi-detail."</p> <p>The default value is Light pink.</p>
Moment-interval stereotype	<p>This parameter controls the color of classifiers with the stereotype "moment-interval."</p> <p>The default value is Light pink.</p>
Party stereotype	<p>This parameter controls the color of classifiers with the stereotype "party."</p> <p>The default value is Light green.</p>
Place stereotype	<p>This parameter controls the color of classifiers with the stereotype "place."</p> <p>The default value is Light green.</p>
Role stereotype	<p>This parameter controls the color of classifiers with the stereotype "role."</p> <p>The default value is Yellow.</p>
Thing stereotype	<p>This parameter controls the color of classifiers with the stereotype "thing."</p> <p>The default value is Light green.</p>
Enable UML in color	<p>This option controls if the color of a classifier depends on the stereotype assigned. For each stereotype, you can select its individual color from the drop-down list (see above).</p> <p>The default value is True.</p>

Together Diagram Layout Options

Layout options define the alignment of diagram elements.

General group

Option	Description and default value
Layout algorithm	<p>UML diagrams can be thought of as graphs (with vertices and edges). Therefore, graph data structures (algorithms) can be applied to the UML diagrams for diagram layout. Click the drop-down arrow to select a layout algorithm. The various algorithms and their optional settings are described below. The algorithm that you specify executes when choosing Layout ▶ Do full layout on the diagram context menu. The following options are available:</p> <ul style="list-style-type: none">• <autoselect>• Hierarchical• Together• Tree• Orthogonal• Spring Embedder <p>The default value is Together.</p>
Recursive layout	<p>This option is available for all layout algorithms. Selecting this option allows to layout all subelements within containers while laying out diagram nodes, thus enabling to lay out inner substructure. This option is useful for the composite states or components.</p>

Hierarchical group

Option	Description and default value
Hybrid proportion parameter	<p>Used in conjunction with the Hybrid ordering heuristic. The optimal setting for this value is 0.7.</p>
Inheritance	<p>This option defines how nodes are aligned with each other if they are connected by an inheritance link.</p> <p>Horizontal - Nodes connected by inheritance are aligned horizontally</p> <p>Vertical - Nodes connected by inheritance are aligned vertically</p>
Justification	<p>This option defines the adjustment of nodes. The Justification setting depends on the Inheritance setting. Select from the following:</p> <ul style="list-style-type: none">• Top: If the Inheritance option is set as Vertical, all nodes in a column are aligned at the left of the column. If the Inheritance option is set as Horizontal, all nodes in a row are aligned at the top of the row.• Center: If the Inheritance option is set as Vertical, all nodes in a column are aligned at the center of the column. If the Inheritance option is set as Horizontal, all nodes in a row are aligned at the center of the row.• Bottom: If the Inheritance option is set as Vertical, all nodes in a column are aligned at the right of the column. If the Inheritance option is set as Horizontal, all nodes in a row are aligned at the bottom of the row.
Layer ordering heuristics	<p>The heuristics are used to sort nodes among each layer to minimize edge-crossings:</p>

- **Barycenter:** The Barycenter heuristic reorders the nodes on node N according to the barycenter weight. The weight of node N is calculated as a simple average of all its successors/predecessors relative coordinates.
- **Median:** The Median heuristic reorders the nodes on node N according to the median weight. The weight of node N is calculated as a simple average of the relative positions of this node dealing only with two central successors/predecessors coordinates.
- **Hybrid:** The Hybrid heuristic combines the Median and Barycenter heuristics.

Minimal horizontal / vertical distance Minimal allowed distance between elements in pixels. Here you can specify Vertical and Horizontal distance options.

Together group

Option	Description and default value																									
Inheritance	<p>This option defines how nodes are aligned with each other if they are connected by an inheritance link. Select either:</p> <ul style="list-style-type: none">• From left to right - nodes connected by inheritance are aligned horizontally from left to right.• From right to left - nodes connected by inheritance are aligned horizontally from right to left.• From top to bottom - nodes connected by inheritance are aligned vertically from top to bottom.• From bottom to top - nodes connected by inheritance are aligned vertically from bottom to top.																									
Justification	<p>This option defines the adjustment of nodes. The Justification setting depends on the Inheritance setting. The elements are aligned as summarized in the following table:</p> <table><tr><td>Inheritance:</td><td>Justification:</td><td>Top:</td><td>Center:</td><td>Bottom:</td></tr><tr><td>Left-right</td><td>Right of the column</td><td>Center of the column</td><td>Left of the column</td><td></td></tr><tr><td>Right-left</td><td>Left of the column</td><td>Center of the column</td><td>Right of the column</td><td></td></tr><tr><td>Top-bottom</td><td>Bottom of the row</td><td>Center of the row</td><td>Top of the row</td><td></td></tr><tr><td>Bottom-top</td><td>Top of the row</td><td>Center of the row</td><td>Bottom of the row</td><td></td></tr></table>	Inheritance:	Justification:	Top:	Center:	Bottom:	Left-right	Right of the column	Center of the column	Left of the column		Right-left	Left of the column	Center of the column	Right of the column		Top-bottom	Bottom of the row	Center of the row	Top of the row		Bottom-top	Top of the row	Center of the row	Bottom of the row	
Inheritance:	Justification:	Top:	Center:	Bottom:																						
Left-right	Right of the column	Center of the column	Left of the column																							
Right-left	Left of the column	Center of the column	Right of the column																							
Top-bottom	Bottom of the row	Center of the row	Top of the row																							
Bottom-top	Top of the row	Center of the row	Bottom of the row																							

Tree group

Option	Description and default value
Hierarchy	<p>This option defines the hierarchy direction of the elements.</p> <ul style="list-style-type: none"> • Horizontal - Elements are aligned horizontally. • Vertical - Elements are aligned vertically.
Reverse hierarchy	The last in the hierarchy element is laid out first in the diagram.
Minimal horizontal (or: vertical) distance	Distance between elements is in pixels. Here you can specify Vertical and Horizontal distance options.
Justification	<p>This option defines the adjustment of elements. The Justification setting is dependent on the Hierarchy direction setting. Select from the following:</p> <ul style="list-style-type: none"> • Top: If the Hierarchy direction option is set to Vertical, all nodes in a column are aligned at the left of the column. If the Hierarchy direction

option is set to Horizontal, all nodes in a row are aligned at the top of the row.

- Center: If the Hierarchy direction option is set to Vertical, all nodes in a column are aligned at the center of the column. If the Hierarchy direction option is set to Horizontal, all nodes in a row are aligned at the center of the row.

- Bottom: If the Hierarchy direction option is set to Vertical, all nodes in a column are aligned at the right of the column. If the Hierarchy direction option is set to Horizontal, all nodes in a row are aligned at the bottom of the row.

Process non-tree edges

If this option is selected, non-tree edges are bent to fit into the diagram layout.

Orthogonal group

Option	Description and default value
Node placement strategy	<p>There are three strategies for node placement: Tree, Balanced, and Smart.</p> <ul style="list-style-type: none">• Tree: The Tree node placement strategy creates a spanning-tree diagram layout. The spanning-tree for the given graph is calculated and diagram nodes are placed on the lattice to minimize the tree edges length. This minimizes the distance between nodes that are linked with a tree-edge.• Balanced: The Balanced node placement strategy uses a balanced ordering of the vertices of the graph as a starting point. Balanced means that the neighbors of each vertex V are as evenly distributed to the left and right of V as possible.• Smart: The Smart node placement strategy sorts all vertices according to the in/out degrees for each vertex and fills the lattice starting from the center with the vertices with the greatest degree.
Distance between elements	Distance is in pixels. Specifies the minimum distance between diagram elements.

Spring Embedder group

Option	Description and default value
Spring force	Specify the rigidity of the springs. The greater value you specify, the less will be the length of edges in the final graph.
Spring movement factor	Specify the nodes movement factor. The more value you specify, the more distance will be between the nodes in the final graph. If you specify 0 as the movement factor, you will get random layout of the nodes.

Together Diagram Print Options

The Print options define default settings that apply to your printed diagrams.

Note that after these settings are applied to the printed material, OS-specific and printer-specific settings will be applied as well.

The tables below list the Print options, descriptions, and default values.

Appearance group

Appearance options:

Option	Description and default value
Print compartments as lines	When this option is true, a control bar displays over the compartments of diagram elements (fields, methods, classes, properties, and so on). The compartment nodes are expanded in the printed results. The default value is False.
Print shadows	When this option is true, a shadow appears under each diagram element in the printed results to create a three-dimensional effect. The default value is True.

Footer group

Option	Description and default value
Footer alignment	This option enables you to select the footer alignment. The default value is Left.
Footer text	This option defines the text that is printed at the bottom of each page, if the Print footer option is true. System macros can be used in the text. See System macros
Print footer	The default value is <code>Printed by %USER% (%LONGDATE%)</code> . If this option is true, the text specified in the Footer text option is printed at the bottom of each page. The default value is True.

General group

General print options:

Option	Description and default value
Fit to page	If this option is true, the Print zoom setting is ignored, and the entire diagram prints on a single page. The default value is False.
Font	This option defines the font (and font size) used in printed diagrams. The default value is <code>Microsoft Sans Serif, 9.75pt</code> .
Print border	If this option is true, a border is printed around the edge of each page. The border corresponds to the Margins settings.

	The default value is True.
Print empty pages	If this option is true, printing includes any blank pages that appear. If this option is false, blank pages are skipped during printing.
	The default value is False.
Print zoom	This option defines the zoom factor for printing diagrams (1:1, 2:1, etc.). Think of the value 1 as being equal to 100%. A value of 2 prints a diagram at 200%, while 0.5 prints it at 50%, and so on. Use the decimal separator as defined in your regional settings.
	The default value is 1.

Header group

Option	Description and default value
Header alignment	This option enables you to select the header alignment. The default value is Left.
Header text	This option defines the text that is printed at the top of each page, if the Print header option is true. System macros can be used in the text. See System macros.
Print Header	The default value is %PROJECT%, %DIAGRAM%. If this option is true, the text specified in the Header text option is printed at the top of each page. The default value is True.

Margins group

Option	Description and default value
Bottom margin	The bottom margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.
Left margin	The left margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.
Right margin	The right margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.
Top margin	The top margin offset in inches. Use the decimal separator as defined in your regional settings. The default value is 1.

Page numbers group

Option	Description and default value
Page number alignment	This option enables you to select the page number alignment. The default value is Left.
Print page numbers	If this option is true, page numbers are printed. The default value is True.

Paper group

Option	Description and default value
Custom paper height (in inches)	<p>This option defines custom paper dimensions for printing. Settings here are only effective if the Paper size option is set to Custom.</p> <p>The default value is 11.88.</p>
Custom paper width (in inches)	<p>This option defines custom paper dimensions for printing. Settings here are only effective if the Paper size option is set to Custom.</p> <p>The default value is 8.4.</p>
Paper orientation	<p>This option defines the orientation of the page. If a Custom paper size is selected, Portrait orientation uses the width and height values specified in the Custom paper height and width settings, while Landscape orientation exchanges the width and height values.</p> <p>The default value is Portrait.</p>
Paper size	<p>This option defines the default paper dimensions for printing. The list of choices includes the most popular paper sizes. If you need to specify your own size, select Custom from the drop down list, and define the dimensions in the Custom paper height and Custom paper width fields.</p> <p>The default value is A4.</p>

Together Diagram View Filters Options

The View Filter group of options provide a set of filters that enable you to control the type of data displayed in different views of a model.

The View Filters options control what elements display on your class and namespace (package) diagrams. The table below lists the filters, descriptions, and default values.

Link filters group

Option	Description and default value
Show association links	This filtering option controls showing/hiding association links in the current project. If it is set to true, association links are displayed. The default value is True.
Show dependency links	This filtering option controls showing/hiding dependency links in the current project. If it is set to true, dependency links are displayed. The default value is True.
Show generalization links	This filtering option controls showing/hiding generalization links in the current project. If it is set to true, generalization links are displayed. The default value is True.
Show implementation links	This filtering option controls showing/hiding implementation links in the current project. If it is set to true, implementation links are displayed. The default value is True.

Member filters group

Option	Description and default value
Show fields	This filtering option controls showing/hiding fields in the current project. If it is set to true, fields are displayed. The default value is True.
Show indexers	This filtering option controls showing/hiding indexers in the current project. If it is set to true, indexers are displayed. The default value is True.
Show members	This filtering option controls showing/hiding members in the current project. If it is set to true, members are displayed. The default value is True.
Show methods	This filtering option controls showing/hiding methods in the current project. If it is set to true, methods are displayed. The default value is True.
Show non public members	This filtering option controls showing/hiding nonpublic members in the current project. If it is set to true, nonpublic members are displayed. The default value is True.
Show properties	This filtering option controls showing/hiding properties in the current project. If it is set to true, properties are displayed. The default value is True.

Node filters group

Option	Description and default value
Show classes	<p>This filtering option controls showing/hiding classes in the current project. If it is set to true, classes are displayed.</p> <p>The default value is True.</p>
Show constraints	<p>This filtering option controls showing/hiding OCL constraints in the current project. If it is set to true, constraints are displayed.</p> <p>The default value is True.</p>
Show delegates	<p>This filtering option controls showing/hiding delegates in the current project. If it is set to true, delegates are displayed.</p> <p>The default value is True.</p>
Show enumerations	<p>This filtering option controls showing/hiding enums in the current project. If it is set to true, enums are displayed.</p> <p>The default value is True.</p>
Show events	<p>This filtering option controls showing/hiding events in the current project. If it is set to true, events are displayed.</p> <p>The default value is True.</p>
Show interfaces	<p>This filtering option controls showing/hiding interfaces in the current project. If it is set to true, interfaces are displayed.</p> <p>The default value is True.</p>
Show modules	<p>This filtering option controls showing/hiding modules in the current project. If it is set to true, modules are displayed.</p> <p>The default value is True.</p>
Show namespaces	<p>This filtering option controls showing/hiding namespaces in the current project. If it is set to true, namespaces are displayed.</p> <p>The default value is True.</p>
Show non public classes	<p>This filtering option controls showing/hiding nonpublic classes in the current project. If it is set to true, nonpublic classes are displayed.</p> <p>The default value is True.</p>
Show notes	<p>This filtering option controls showing/hiding notes in the current project. If it is set to true, notes are displayed.</p> <p>The default value is True.</p>
Show shortcuts	<p>This filtering option controls showing/hiding shortcuts in the current project. If it is set to true, shortcuts are displayed.</p> <p>The default value is True.</p>
Show structures	<p>This filtering option controls showing/hiding structures in the current project. If it is set to true, structures are displayed.</p> <p>The default value is True.</p>

Together Generate Documentation Options

The Generate Documentation options control the variety of content (as well as appearance) to include or exclude from your generated HTML documentation. The table below lists the Generate Documentation options, descriptions, and default values.

General group

Option	Description and default value
Bottom	Specifies the text to be placed at the bottom of each output file. The text will be placed at the bottom of the page, below the lower navigation bar. The text can contain HTML tags and white spaces.
Documentation Title	Specifies the title to be placed at the top of the overview summary file. The title will be placed as a centered, level-one heading directly beneath the upper navigation bar. It can contain HTML tags and white spaces.
Footer	Specifies the footer text to be placed at the bottom of each output file. The footer is placed to the right of the lower navigation bar. It can contain HTML tags and white spaces.
Header	Specifies the header text to be placed at the top of each output file. The header is placed to the right of the upper navigation bar. It can contain HTML tags and white spaces.
Navigation type	Specifies the location for descriptions of design elements: Package: in package files Diagram: in diagram files The default value is Package.
Use Internal Browser	When this options is true, an internal browser is used for documentation presentation. The default value is False.
Window Title	Specifies the title to be placed in the HTML <code><title></code> tag. This text appears in the window title and in any browser bookmarks (favorites) created for this page. This title should not contain any HTML tags, since the browser does not interpret them properly.

Include group

Option	Description and default value
internal (package)	If this option is true, internal classes, interfaces and members are shown in the generated documentation. The default value is True.
private	If this option is true, private classes, interfaces and members are shown in the generated documentation. The default value is False.
protected	If this option is true, protected classes, interfaces and members are shown in the generated documentation. The default value is False.
protected internal	If this option is true, protected internal classes, interfaces and members are shown in the generated documentation. The default value is False.
public	If this option is true, public classes, interfaces and members are shown in the generated documentation.

The default value is True.

Navigation group

Option	Description and default value
Generate Help	<p>This option controls whether to put the HELP link in the navigation bars at the top and bottom of each page of output.</p> <p>The default value is True.</p>
Generate Index	<p>This option controls whether to generate the index.</p> <p>The default value is True.</p>
Generate Navbar	<p>This option controls whether to generate the navigation bar, header, and footer, otherwise found at the top and bottom of the generated pages.</p> <p>The default value is True.</p>
Generate Tree	<p>This option controls whether to generate the class/interface hierarchy.</p> <p>The default value is True.</p>
Generate Use	<p>If this option is true, one Use page is included for each documented class and namespace. The page describes which namespaces, classes, methods, constructors, and fields use any API of the given class or namespace. Given class C, things that use class C would include subclasses of C, fields declared as C, methods that return C, and methods and constructors with parameters of type C.</p> <p>The default value is False.</p>

Together Model View Options

The Model View options control how diagram content displays in the **Model View**. The table below lists the Model View general options, descriptions, and default values.

Option	Description and default value
Show diagram nodes expandable	If this option is set to true, the Model View displays expandable diagram nodes with the elements contained therein. The default value is True.
Show links	If this option is set to true, the Model View displays links between nodes. Otherwise the links are hidden. The default value is False.
Sorting type	This option enables you to select the type of sorting for the Model View (alphabetical, by metaclass, or none). The default value is Metaclass.
View type	This option controls the type of presentation of the Model. Diagram-centric mode assumes that the design elements are shown in their respective diagrams, and only the classes, interfaces and the other source-code elements are shown in the namespaces. Model-centric mode assumes that all elements are shown under namespaces. The default value is Diagram-centric.

Together Source Code Options

The Source Code options allows you to control whether dependency links are drawn automatically on diagrams. The table below lists the Source Code general options, descriptions, and default values.

Option	Description and default value
Autocreate association links derived from properties	Set this option to True to create association links for properties automatically, while parsing the source code. The default value is False.
Autocreate dependency links	If the option is true, the dependency links are drawn automatically on diagrams. The default value is False.
Automatically maintain namespace folder structure	If this parameter is true, new classes will be created in subfolders. The structure of these subfolders will be detected using the existing folder structure, if any. For new namespaces, new subfolders will be created. If this parameter is false, new classes will be created in the project root folder. The default value is True.
Encoding	This parameter specifies the character encoding to be applied to source code files. System default lets you use the current OS encoding. The default value is system default.

UML 1.5 Reference

This section contains reference material about UML 1.5 diagrams.

UML 1.5 Class Diagrams

This section describes the elements of UML 1.5 Class Diagrams.

In This Section

[UML 1.5 Class Diagram Definition](#)

Provides UML 1.5 class diagram definition.

[Class Diagram Types](#)

Describes Class diagram types: logical and package (namespace) diagrams.

[UML 1.5 Class Diagram Elements](#)

Provides UML 1.5 class diagram elements.

[Class Diagram Relationships](#)

Describes class diagram relationships.

[LiveSource Rules](#)

Describes LiveSource rules.

[Association Class and N-ary Association](#)

Describes association class and n-ary association

[Inner Classifiers](#)

Describes possible inner classifiers on Class diagrams.

[Members](#)

Describes possible members of classifiers.

UML 1.5 Class Diagram Definition

Using Together, you can create language-neutral class diagrams in design projects, or language-specific class diagrams in implementation projects. For implementation projects, all diagram elements are immediately synchronized with the source code.

Definition

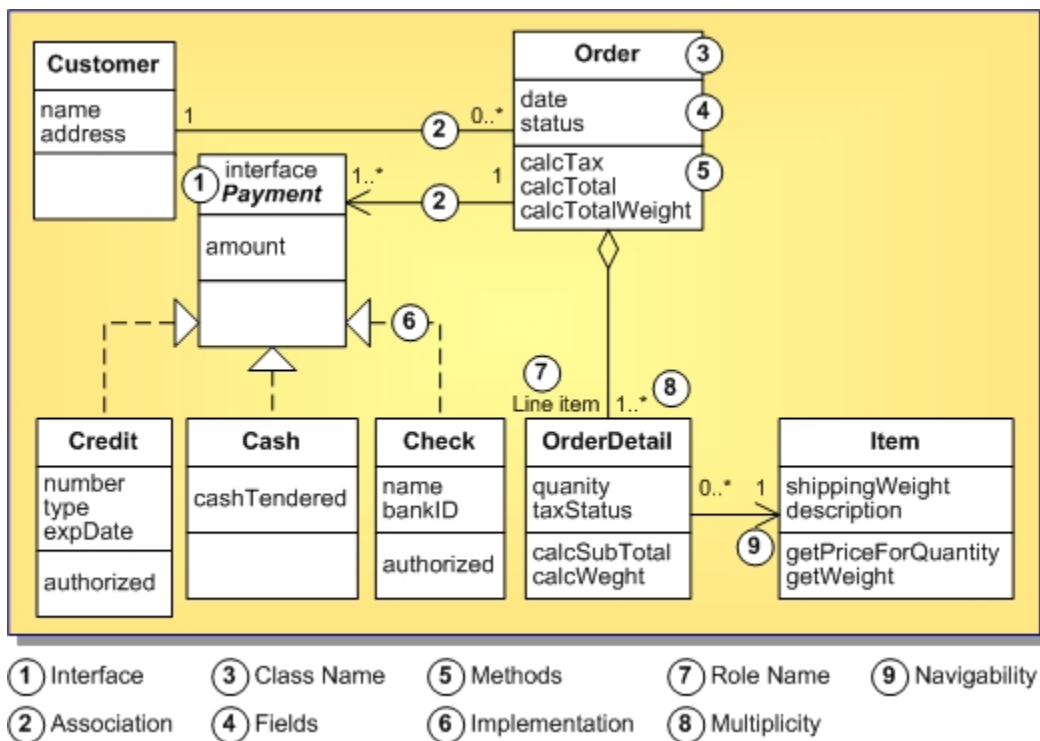
A class diagram provides an overview of a system by showing its classes and the relationships among them. Class diagrams are static: they display what interacts but not what happens during the interaction.

UML class notation is a rectangle divided into three parts: class name, fields, and methods. Names of abstract classes and interfaces are in italics. Relationships between classes are the connecting links.

In Together, the rectangle is further divided with separate partitions for properties and inner classes.

Sample Diagram

The following class diagram models a customer order from a retail catalog. The central class is the *Order*. Associated with it are the *Customer* making the purchase and the *Payment*. There are three types of payments: *Cash*, *Check*, or *Credit*. The order contains *OrderDetails* (line items), each with its associated *Item*.



There are three kinds of relationships used in this example:

- Association: For example, an *OrderDetail* is a line item of each *Order*.
- Aggregation: In this diagram, *Order* has a collection of *OrderDetails*.
- Implementation: *Payment* is an interface for *Cash*, *Check*, and *Credit*.

Class Diagram Types

There are two types of class diagrams used in Together:

- **Package (namespace)** diagrams. These diagrams are referred to as package diagrams in design projects, and namespace diagrams in implementation projects. They are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project group with the file extension `.txvpck`.
- **Logical** class diagrams. These diagrams are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project group with the file extension `.txvcls`.

Together automatically creates a default namespace diagram for the project and for each subdirectory under the project directory. The default project diagram is named default; the default namespace (package) diagrams are named after the respective namespaces (packages).

You create logical class diagrams manually by using the **Add ► Class Diagram** or **Add ► Other Diagram** command on the project context menu.

UML 1.5 Class Diagram Elements

The table below lists the elements of UML 1.5 class diagrams that are available using the **Tool Palette**.

UML 1.5 class diagram elements

Name	Type
Namespace (Package)	node
Class	node
Interface	node
Association Class	node
Structure	node
Enumeration	node
Delegate	node
Object	node
Generalization/Implementation	link
Association	link
Dependency	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Constraint	OCL node
Constraint link	OCL link
Note	annotation
Note Link	annotation link

Class Diagram Relationships

There are several kinds of relationships:

- **Association:** A relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes. Associations can be directed or undirected. A directed link points to the supplier class (the target). An association has two ends. An end may have a role name to clarify the nature of the association. A navigation arrow on an association shows which direction the association can be traversed or queried. A class can be queried about its Item, but not the other way around. The arrow also lets you know who "owns" the implementation of the association. Associations with no navigation arrows are bi-directional.
- **Generalization/Implementation:** An inheritance link indicating that a class implements an interface. An implementation has a triangle pointing to the interface.
- **Dependency**

There are several subtypes of an association relationship:

- **Simple Association**
- **Aggregation:** An association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole.
- **Composition**

Every class diagram has classes and associations. Navigability, roles, and multiplicities are optional items placed in a diagram to provide clarity.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers.

This table lists the most common multiplicities:

Multiplicities

Multiplicity	Meaning
0..1	Zero or one instance. The notation n . . m indicates n to m instances
0..* or *	No limit on the number of instances (including none)
1	Exactly one instance
1..*	At least one instance

LiveSource Rules

The impact of changing a class, interface, or namespace on a logical class diagram varies according to the kind of change:

- Changing the name, adding a member, creating a new link, or applying a pattern makes the corresponding change in the actual source code.
- Choose **Delete from View** on the context menu of the element to remove the element from a current diagram and keep the element in the namespace (package).
- Choose **Delete** on the context menu to completely remove the element from the model.
- When you press `Delete` on the keyboard, the **Delete from view** command is applied, if it is available in this particular situation. If it is not, the element is deleted completely.
- Direct changes in source code editor, such as renaming a class, cannot be tracked by Together. Use refactoring operations for this purpose.

Association Class and N-ary Association

Association classes appear in diagrams as three related elements:

- Association class itself (represented by a class icon)
- N-ary association class link (represented by a diamond)
- Association connector (represented by a link between both)

Association classes can connect to as many association end classes (participants) as required.

The **Object Inspector** of an association class, association link, and connector contain an additional Association tab. This tab contains the only label property, its value being synchronized with the name of the association class. For the association classes and association end links, the Custom node of the **Object Inspector** displays additional properties that corresponds to the role of this part of n-ary association ([associationClass](#) and [associationEnd](#) respectively).

You can delete each of the association end links or participant classes without destroying the entire n-ary association. However, deleting the association class results in deleting all the components of the n-ary association.

Inner Classifiers

The table below lists the diagram container elements along with the inner classifiers that you can add to container elements.

Inner classifiers

Container element	Inner classifiers available
Class	Class
	Interface
	Structure [C#]
	Delegate [C#]
	Delegate as Function
	Enum [C#]
Interface	Class]
	Interface
	Delegate
	Delegate as Function
	Enum
Structure	Class
	Interface
	Structure
	Delegate
	Delegate as
	Function
	Enum
Module	Class
	Interface
	Structure
	Delegate
	Delegate as
	Function
	Enum

Members

Note that the set of available members is different for the design and implementation projects.

The table below lists the diagram elements along with the members that can be added using the context menu of the element. The type of applicable project is specified in square brackets.

Members available

Container element	Members available
Class	Method [C#]
	Operation [Design]
	Constructor [Design, C#]
	Destructor [C#]
	Field [C#]
	Attribute [Design]
	Property [C#]
	Indexer [C#]
	Event [C#]
Interface	Method [C#]
	Property [C#]
	Indexer [C#]
	Event [C#]
	Attribute [Design]
	Operation [Design]
Structure	Method
	Constructor
	Field
	Property
	Indexer
	Event
Module	Function
	Subroutine
	Constructor
	Field
	Property
Enumeration	Enum Value

For implementation projects: If you set the abstract property for a method, property, or indexer (in abstract classes) as True in the Properties Window, the method body is removed from the source code. This is the desired behavior. Resetting the abstract property to False in the **Object Inspector**, adds a new empty method body.

UML 1.5 Use Case Diagrams

This section describes the elements of UML 1.5 Use Case Diagrams.

In This Section

[UML 1.5 Use Case Diagram Definition](#)

Provides UML 1.5 use case diagram definition.

[UML 1.5 Use Case Diagram Elements](#)

Describes UML 1.5 use case diagram elements.

[Extension Point](#)

Describes an extension point (Use Case diagrams).

UML 1.5 Use Case Diagram Definition

Use case diagrams are helpful in three areas:

- Determining features (requirements): New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients: Notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- Generating test cases: The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Definition

Use Case Diagrams describe what a system does from the viewpoint of an external observer. The emphasis is on what a system does rather than how.

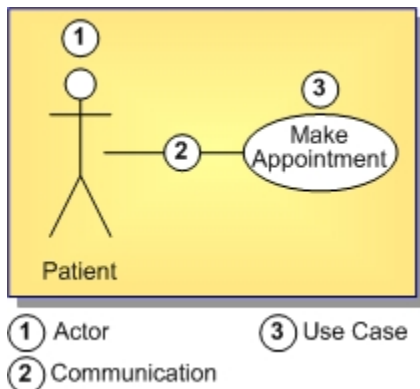
Use Case Diagrams are closely connected to scenarios. A scenario is an example of what happens when someone interacts with the system.

Sample Diagram

Following is a scenario for a medical clinic:

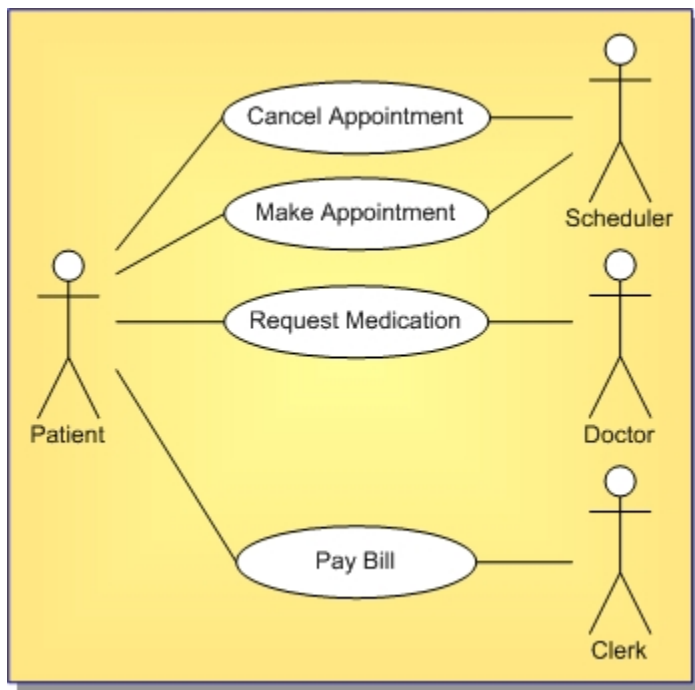
A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot.

A use case is a summary of scenarios for a single task or goal. An actor is who or what initiates the events involved in that task. Actors are simply roles that people or objects play. The following diagram is the [Make Appointment](#) use case for the medical clinic. The actor is a [Patient](#). The connection between actor and use case is a communication association (or communication for short).



Actors are stick figures. Use cases are ovals. Communications are lines that link actors to use cases.

A use case diagram is a collection of actors, use cases, and their communications. Following is an example of the use case Make Appointment as part of a diagram with four actors and four use cases. Notice that a single use case can have multiple actors.



UML 1.5 Use Case Diagram Elements

The table below lists the elements of UML 1.5 Use Case diagrams that are available using the **Tool Palette**.

UML 1.5 Use Case diagram elements

Name	Type
Actor	node
Use Case	node
Communicates	link
Extend	link
Include	link
Generalization	link
System Boundary	node
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

Extension Point

An extension point refers to a location within a use case where you can insert action sequences from other use cases.

An extension point consists of a unique name within a use case and a description of the location within the behavior of the use case.

In a use case diagram, extension points are listed in the use case with the heading "Extension Points" (appears as bold text in the Diagram View).

UML 1.5 Interaction Diagrams

This section describes the elements of UML 1.5 Sequence and Collaboration diagrams.

In This Section

[UML 1.5 Sequence Diagram Definition](#)

Provides UML 1.5 sequence diagram definition.

[UML 1.5 Collaboration Diagram Definition](#)

Provides UML 1.5 collaboration diagram definition.

[UML 1.5 Interaction Diagram Elements](#)

Describes UML 1.5 interaction diagram elements.

[Conditional Block](#)

Describes a conditional block.

[UML 1.5 Message](#)

Describes UML 1.5 messages.

[Activation Bar](#)

Describes an activation bar.

[Nested Message](#)

Describes a nested message.

UML 1.5 Sequence Diagram Definition

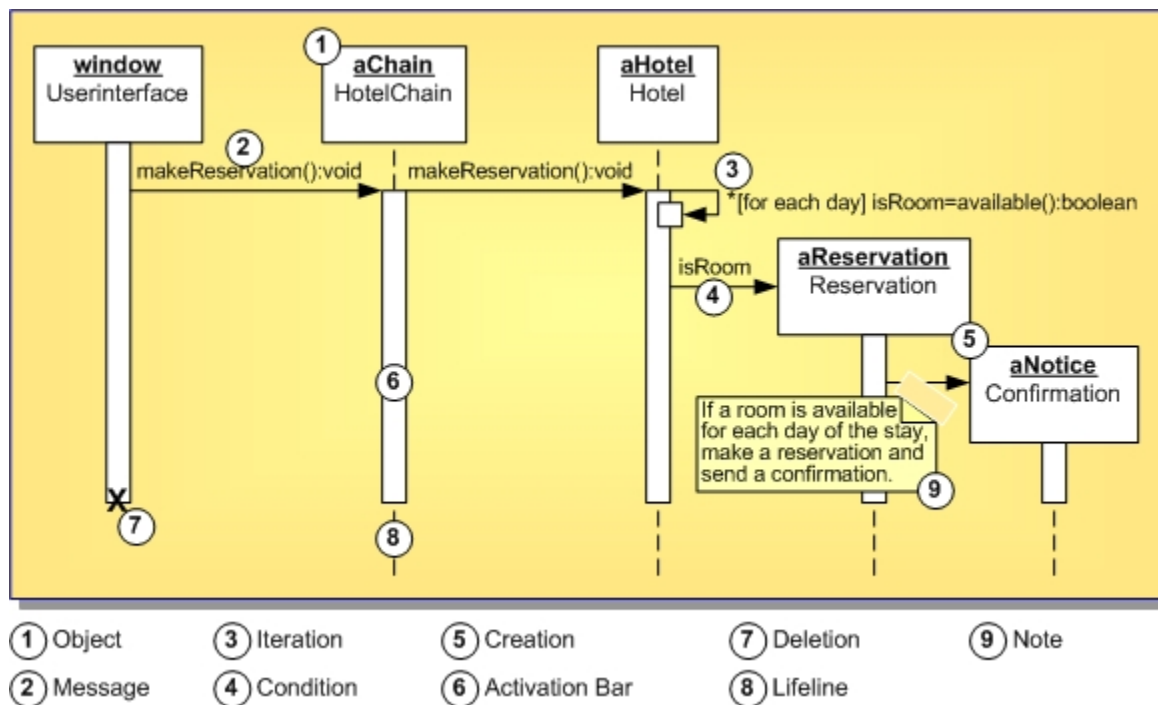
Class diagrams are static model views. In contrast, interaction diagrams are dynamic, describing how objects collaborate.

Definition

A sequence diagram is an interaction diagram that details how operations are carried out: what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.

Sample Diagram

Following is a Sequence Diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window (the `UserInterface`).



The `UserInterface` sends a `makeReservation()` message to a `HotelChain`. The `HotelChain` then sends a `makeReservation()` message to a `Hotel`. If the `Hotel` has available rooms, then it makes a `Reservation` and a `Confirmation`.

Each vertical dotted line is a lifeline, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the activation bar of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message.

In this diagram, the `Hotel` issues a self call to determine if a room is available. If so, then the `Hotel` creates a `Reservation` and a `Confirmation`. The asterisk on the self call means iteration (to make sure there is available room for each day of the stay in the hotel). The expression in square brackets, `[]`, is a condition.

The diagram has a clarifying note, which is text inside a dog-eared rectangle. Notes can be included in any kind of UML diagram.

UML 1.5 Collaboration Diagram Definition

Class diagrams are static model views. In contrast, interaction diagrams are dynamic, describing how objects collaborate.

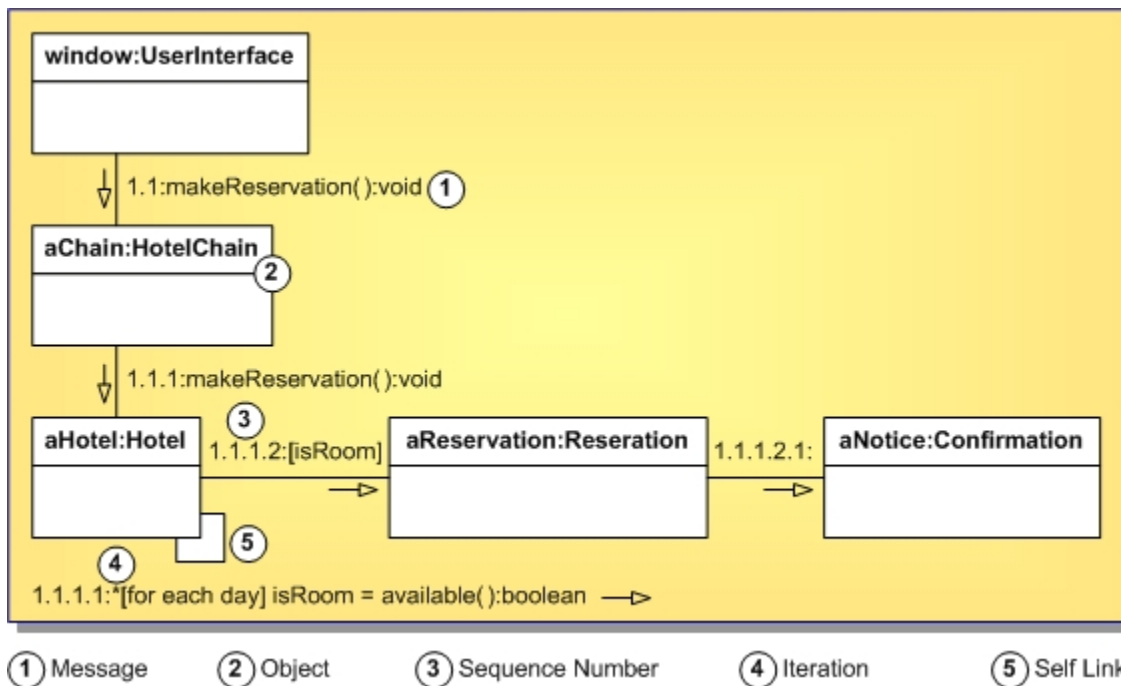
Definition

Like sequence diagrams, collaboration diagrams are also interaction diagrams. Collaboration diagrams convey the same information as sequence diagrams, but focus on object roles instead of the times that messages are sent.

In a sequence diagram, object roles are the vertices and messages are the connecting links. In a collaboration diagram, as follows, the object-role rectangles are labeled with either class or object names (or both). Colons precede the class names (:).

Sample Diagram

Each message in a collaboration diagram has a sequence number. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.



UML 1.5 Interaction Diagram Elements

The table below lists the elements of UML 1.5 Interaction (Sequence and Collaboration) diagrams that are available using the **Tool Palette**.

UML 1.5 interaction diagram elements

Name	Type
Object	node
Actor	node
Message	link
Self Message	link
Message with delivery time	link, Sequence only
Conditional Block	node, Sequence only
Return	link, Sequence only
Association	link, Collaboration only
Aggregates	link, Collaboration only
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

You can add shortcuts to the interaction diagrams, by using the **Add ▶ Shortcut** command. However, referring to the elements of the other interaction diagrams is not allowed.

Conditional Block

Conditional block statement is a flexible tool to enhance a sequence diagram. The following statements are supported:

- if
- else
- for
- foreach
- while
- do while
- try
- catch
- finally
- switch
- case
- default

UML 1.5 Message

By default, message links in a sequence diagram are numbered sequentially from top to bottom. You can reorder messages.

A “self message” is a message from an object back to itself.

Activation Bar

Together automatically renders the activation of messages that show the period of time that the message is active. When you draw a message link to the destination object, the activation bar is created automatically.

You can extend or reduce the period of time of a message by vertically dragging the top or bottom line of the activation bar as required. A longer activation bar means a longer time period when the message is active.

Nested Message

You can nest messages by originating message links from an activation bar. The nested message inherits the numbering of the parent message.

For example, if the parent message has the number 1, its first nested message is 1.1. It is also possible to create message links back to the parent activation bars.

UML 1.5 Statechart Diagrams

This section describes the elements of UML 1.5 Statechart diagrams.

In This Section

[UML 1.5 Statechart Diagram Definition](#)

Provides UML 1.5 statechart diagram definition.

[UML 1.5 Statechart Diagram Elements](#)

Describes UML 1.5 statechart diagram elements.

[State](#)

Describes a state (UML 1.5 Activity, UML 1.5 Statechart, UML 2.0 State Machine diagrams).

[Transition](#)

Describes a transition (UML 1.5 Activity, UML 1.5 Statechart, UML 2.0 State Machine diagrams).

[Deferred Event](#)

Describes a deferred event.

UML 1.5 Statechart Diagram Definition

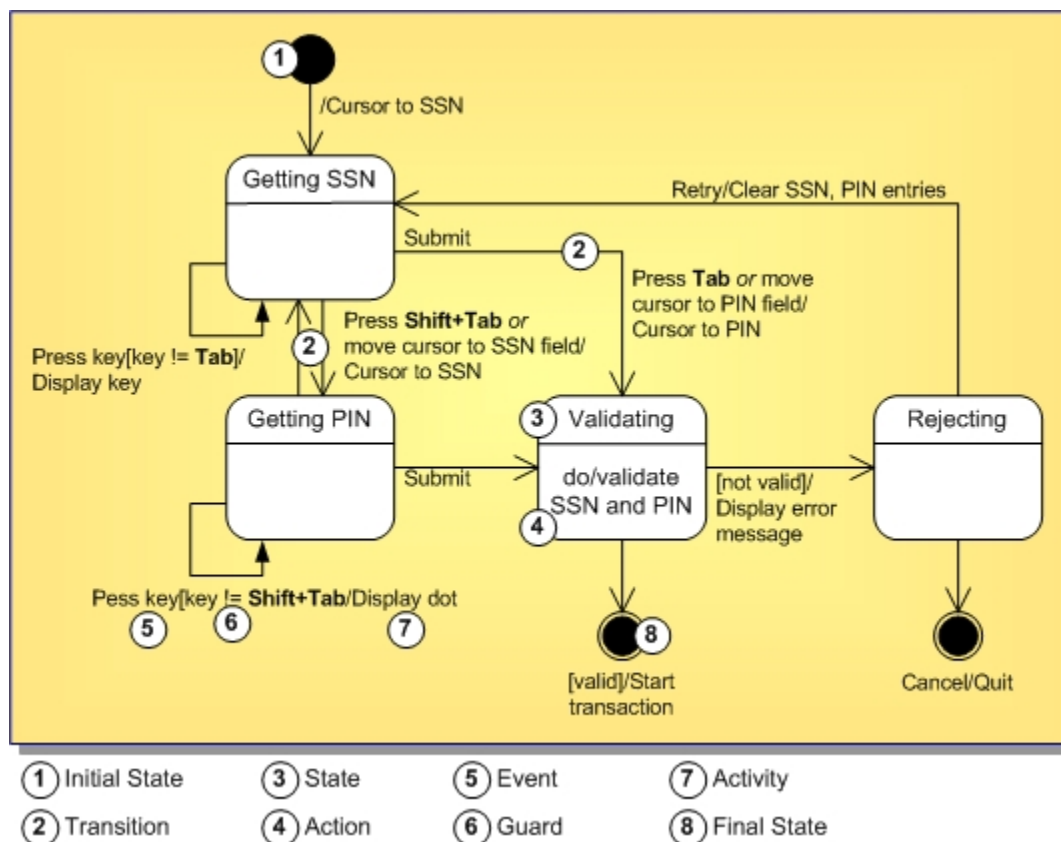
This topic describes the UML 1.5 Statechart Diagram.

Definition

Objects have behaviors and states. The state of an object depends on its current activity or condition. A statechart diagram shows the possible states of the object and the transitions that cause a change in state.

Sample Diagram

The following diagram models the login part of an online banking system. Logging in consists of entering a valid social security number and personal id number, then submitting the information for validation. Logging in can be factored into four non-overlapping states: *Getting SSN*, *Getting PIN*, *Validating*, and *Rejecting*. Each state provides a complete set of transitions that determines the subsequent state.



States are depicted as rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written next to the arrows. This diagram has two self-transitions: *Getting SSN* and *Getting PIN*. The initial state (shown as a black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form */action*. While in its *Validating* state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

UML 1.5 Statechart Diagram Elements

The table below lists the elements of UML 1.5 Statechart diagrams that are available using the **Tool Palette**.

UML 1.5 Statechart diagram elements

Name	Type
State	node
Start State	node
End State	node
History	node
Object	node
Transition	link
Horizontal Fork/Join	node
Vertical Fork/Join	node
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

State

A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (for example, the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).

Actions

Entry and exit actions are executed when entering or leaving a state, respectively.

You can create these actions in statechart diagrams as special nodes, or as stereotyped internal transitions.

Composite (nested) state

Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.

Transition

A single transition comes out of each state or activity, connecting it to the next state or activity.

A transition takes operation from one state to another and represents the response to a particular event. You can connect states with transitions and create internal transitions within states.

Internal transition

An internal transition is a way to handle events without leaving a state (or activity) and dispatching its exit or entry actions. You can add an internal transition to a state or activity element.

An internal transition is shorthand for handling events without leaving a state and dispatching its exit or entry actions.

Self-transition

A self-transition flow leaves the state (or activity) dispatching any exit action(s), then reenters the state dispatching any entry action(s). You can draw self-transitions for both activity and state elements on an activity diagram.

Self-transition for statechart diagrams

Self-transition for activity diagrams

Multiple transition

A transition can branch into two or more mutually-exclusive transitions.

A transition may fork into two or more parallel activities. A solid bar indicates a fork and the subsequent join of the threads coming out of the fork.

A transition may have multiple sources (a join from several concurrent states) or it may have multiple targets (a fork to several concurrent states).

You can show multiple transitions with either a vertical or horizontal orientation in your State and Activity diagrams. Both the State and Activity diagram toolbars provide separate horizontal and vertical fork/join buttons for each orientation. The two orientations are semantically identical.

Guard expressions

All transitions, including internal ones, are provided with the guard conditions (logical expressions) that define whether this transition should be performed. Also you can associate a transition with an effect, which is an optional activity performed when the transition fires. The guard condition is enclosed in the brackets (for example, "[false]") and displayed near the transition link on a diagram. Effect activity is displayed next to the guard condition. You can define the guard condition and effect using the **Object Inspector**.

Guard expressions (inside []) label the transitions coming out of a branch. The hollow diamond indicates a branch and its subsequent merge that indicates the end of the branch.

Deferred Event

A deferred event is like an internal transition that handles the event and places it in an internal queue until it is used or discarded.

A deferred event may be thought of as an internal transition that handles the event and places it in an internal queue until it is used or discarded. You can add a deferred event to a state or activity element.

UML 1.5 Activity Diagrams

This section describes the elements of UML 1.5 Activity Diagrams.

In This Section

[UML 1.5 Activity Diagram Definition](#)

Provides UML 1.5 activity diagram definition.

[UML 1.5 Activity Diagram Elements](#)

Describes UML 1.5 activity diagram elements.

[State](#)

Describes a state (UML 1.5 Activity, UML 1.5 Statechart, UML 2.0 State Machine diagrams).

[Transition](#)

Describes a transition (UML 1.5 Activity, UML 1.5 Statechart, UML 2.0 State Machine diagrams).

[Deferred Event](#)

Describes a deferred event.

UML 1.5 Activity Diagram Elements

The table below lists the elements of UML 1.5 Activity diagrams that are available using the **Tool Palette**.

UML 1.5 activity diagram elements

Name	Type
Activity	node
Decision	node
Signal Receipt	node
Signal Sending	node
State	node
Start State	node
End State	node
History	node
Object	node
Transition	link
Horizontal Fork/Join	node
Vertical Fork/Join	node
Swimlane	node
Object Flow	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

UML 1.5 Activity Diagram Definition

This topic describes the UML 1.5 Activity Diagram.

Definition

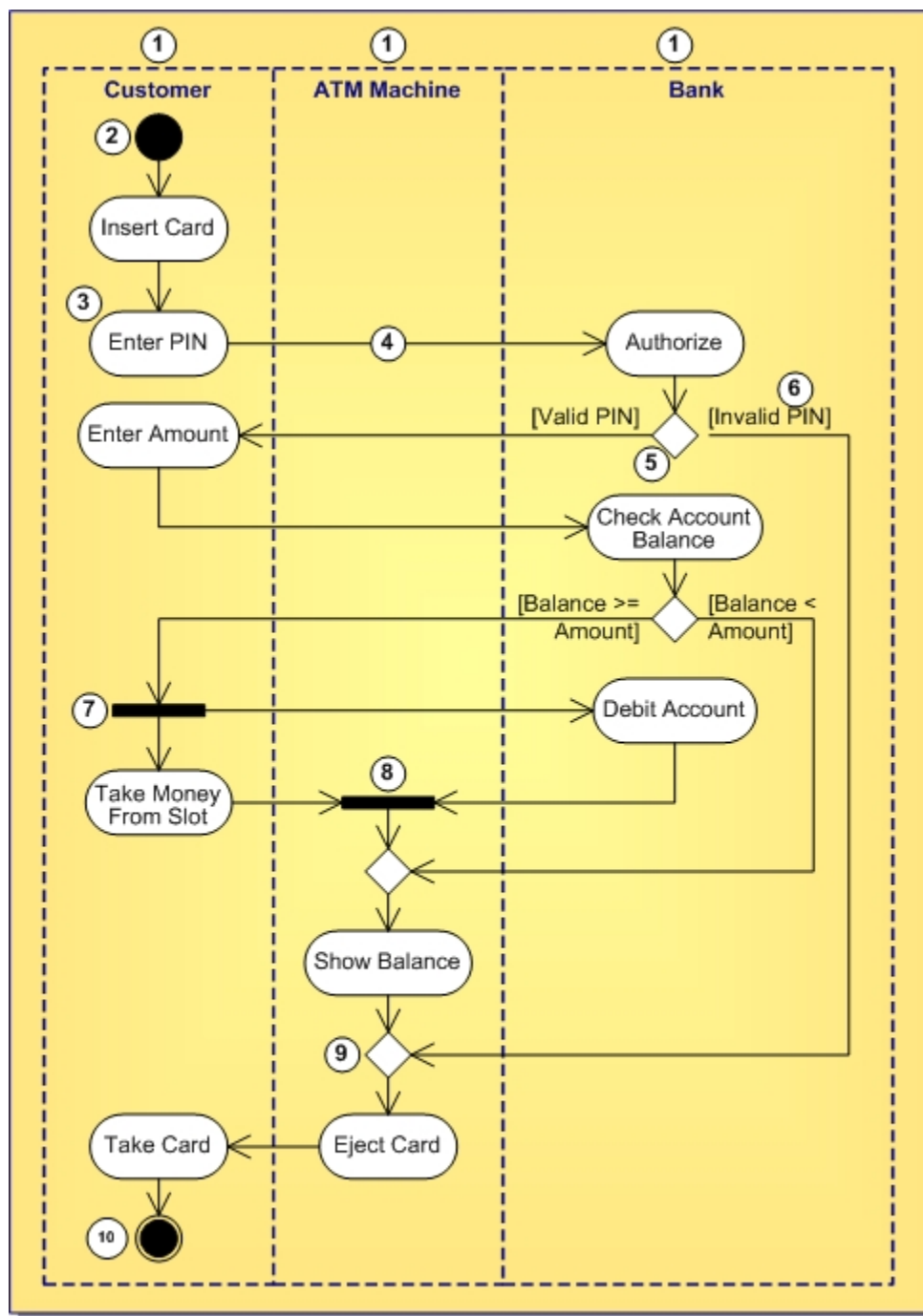
An Activity diagram is similar to a flowchart. Activity diagrams and Statechart diagrams are related. While a Statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an Activity diagram focuses on the flow of activities involved in a single process. The Activity diagram shows how these single-process activities depend on one another.

Activity diagrams can be divided into object swimlanes that determine which object is responsible for an activity.

Sample Diagram

The Activity Diagram below uses the following process: "Withdraw money from a bank account through an ATM." The three involved classes of the activity are Customer, ATM Machine, and Bank. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are shown as rounded rectangles.

The three involved classes (people, and so on) of the activity are Customer, ATM, and Bank. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are shown as rounded rectangles.



- | | | | | |
|------------|---------------|--------------------|--------|---------|
| ① Swimlane | ③ Activity | ⑤ Branch | ⑦ Fork | ⑨ Merge |
| ② Start | ④ Transisiton | ⑥ Guard Expression | ⑧ Join | ⑩ End |

State

A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (for example, the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).

Actions

Entry and exit actions are executed when entering or leaving a state, respectively.

You can create these actions in statechart diagrams as special nodes, or as stereotyped internal transitions.

Composite (nested) state

Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.

Transition

A single transition comes out of each state or activity, connecting it to the next state or activity.

A transition takes operation from one state to another and represents the response to a particular event. You can connect states with transitions and create internal transitions within states.

Internal transition

An internal transition is a way to handle events without leaving a state (or activity) and dispatching its exit or entry actions. You can add an internal transition to a state or activity element.

An internal transition is shorthand for handling events without leaving a state and dispatching its exit or entry actions.

Self-transition

A self-transition flow leaves the state (or activity) dispatching any exit action(s), then reenters the state dispatching any entry action(s). You can draw self-transitions for both activity and state elements on an activity diagram.

Self-transition for statechart diagrams

Self-transition for activity diagrams

Multiple transition

A transition can branch into two or more mutually-exclusive transitions.

A transition may fork into two or more parallel activities. A solid bar indicates a fork and the subsequent join of the threads coming out of the fork.

A transition may have multiple sources (a join from several concurrent states) or it may have multiple targets (a fork to several concurrent states).

You can show multiple transitions with either a vertical or horizontal orientation in your State and Activity diagrams. Both the State and Activity diagram toolbars provide separate horizontal and vertical fork/join buttons for each orientation. The two orientations are semantically identical.

Guard expressions

All transitions, including internal ones, are provided with the guard conditions (logical expressions) that define whether this transition should be performed. Also you can associate a transition with an effect, which is an optional activity performed when the transition fires. The guard condition is enclosed in the brackets (for example, "[false]") and displayed near the transition link on a diagram. Effect activity is displayed next to the guard condition. You can define the guard condition and effect using the **Object Inspector**.

Guard expressions (inside []) label the transitions coming out of a branch. The hollow diamond indicates a branch and its subsequent merge that indicates the end of the branch.

Deferred Event

A deferred event is like an internal transition that handles the event and places it in an internal queue until it is used or discarded.

A deferred event may be thought of as an internal transition that handles the event and places it in an internal queue until it is used or discarded. You can add a deferred event to a state or activity element.

UML 1.5 Component Diagrams

This section describes the elements of UML 1.5 Component Diagrams.

In This Section

[UML 1.5 Component Diagram Definition](#)

Provides UML 1.5 component diagram definition.

[UML 1.5 Component Diagram Elements](#)

Describes UML 1.5 component diagram elements.

UML 1.5 Component Diagram Definition

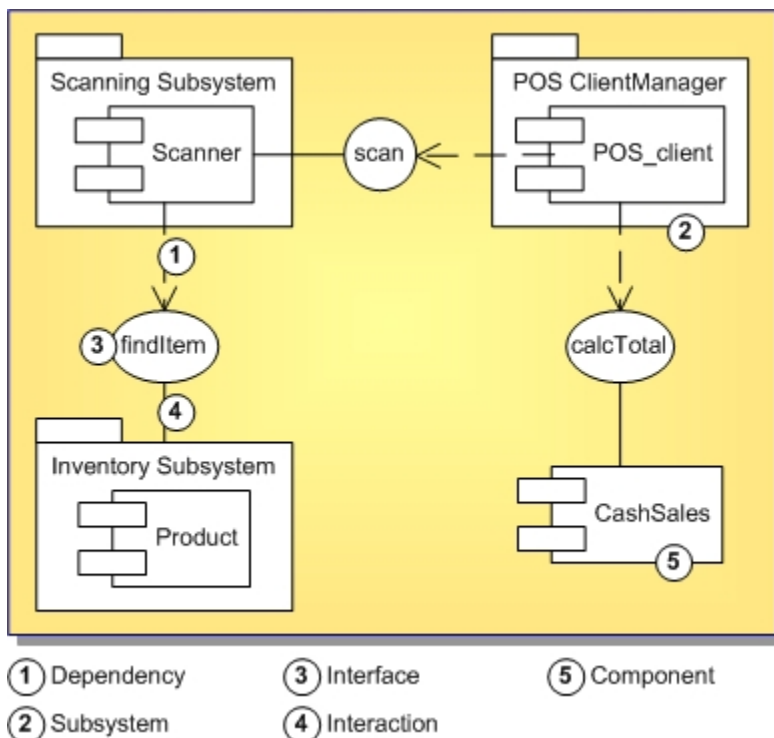
Both component and deployment diagrams depict the physical architecture of a computer-based system. Component diagrams show the dependencies and interactions between software components.

Definition

A component is a container of logical elements and represents things that participate in the execution of a system. Components also use the services of other components through one of its interfaces. Components are typically used to visualize logical packages of source code (work product components), binary code (deployment components), or executable files (execution components).

Sample Diagram

Following is a component diagram that shows the dependencies and interactions between software components for a cash register program.



UML 1.5 Component Diagram Elements

The table below lists the elements of UML 1.5 component diagrams that are available using the **Tool Palette**.

UML 1.5 component diagram elements

Name	Type
Subsystem	node
Component	node
Interface	node
Supports	link
Dependency	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

UML 1.5 Deployment Diagrams

This section describes the elements of UML 1.5 Deployment Diagrams.

In This Section

[UML 1.5 Deployment Diagram Definition](#)

Provides UML 1.5 deployment diagram definition.

[UML 1.5 Deployment Diagram Elements](#)

Describes UML 1.5 deployment diagram elements.

UML 1.5 Deployment Diagram Definition

Both component and deployment diagrams depict the physical architecture of a computer-based system.

Deployment Diagrams are made up of a graph of nodes connected by communication associations to show the physical configuration of the software and hardware.

Components are physical units of packaging in software, including:

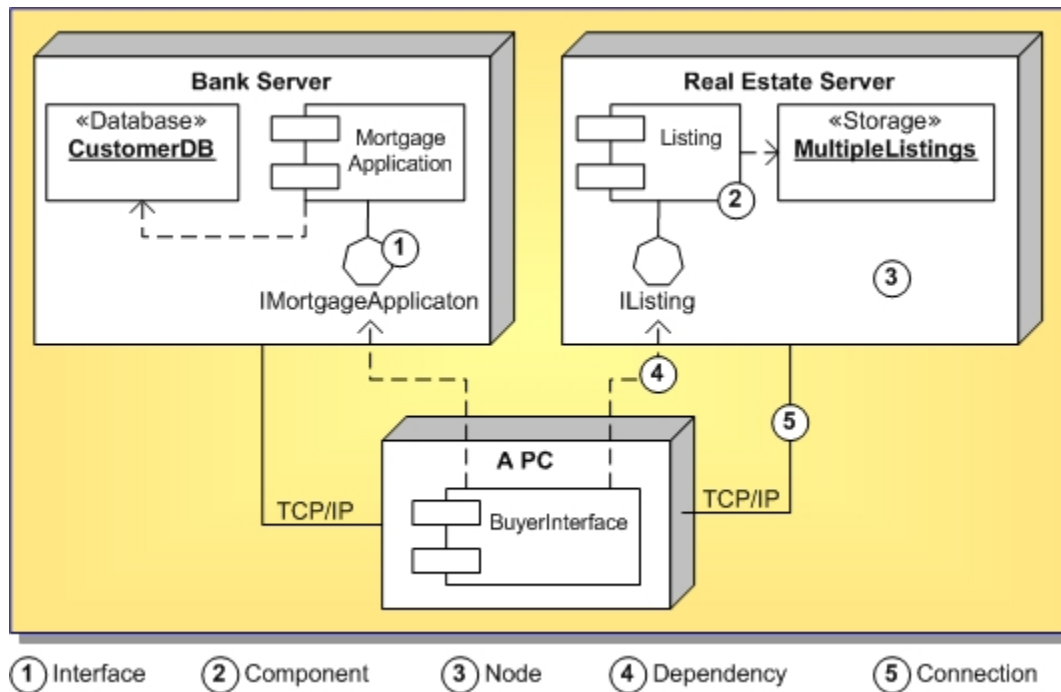
- External libraries
- Operating systems
- Virtual machines

Definition

The physical hardware is made up of nodes. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.

Sample Diagram

Following is a deployment diagram that shows the relationships of software and hardware components for a real estate transaction.



UML 1.5 Deployment Diagram Elements

The table below lists the elements of UML 1.5 deployment diagrams that are available using the **Tool Palette**.

UML 1.5 deployment diagram elements

Name	Type
Node	node
Component	node
Interface	node
Supports	link
Aggregates	link
Object	node
Association	link
Dependency	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

UML 2.0 Reference

This section contains reference material about UML 2.0 diagrams.

UML 2.0 Class Diagrams

This section describes the elements of UML 2.0 Class diagrams.

In This Section

[UML 2.0 Class Diagram Definition](#)

Provides UML 2.0 class diagram definition.

[Class Diagram Types](#)

Describes Class diagram types: logical and package (namespace) diagrams.

[UML 2.0 Class Diagram Elements](#)

Describes UML 2.0 class diagram elements.

[Class Diagram Relationships](#)

Describes class diagram relationships.

[Association Class and N-ary Association](#)

Describes association class and n-ary association

[Inner Classifiers](#)

Describes possible inner classifiers on Class diagrams.

[Members](#)

Describes possible members of classifiers.

UML 2.0 Class Diagram Definition

UML 2.0 Class diagrams feature the same capabilities as the UML 1.5 diagrams.

The UML 2.0 class diagrams offer new diagram elements such as ports, provided and required interfaces, and slots.

According to the UML 2.0 specification, an instance specification can instantiate one or more classifiers. You can use classes, interfaces, or components as a classifier.

Interfaces

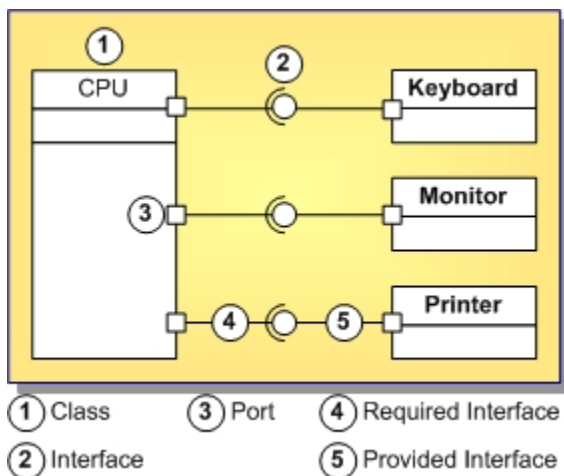
A class implements an interface via the same generalization/implementation link, as in UML 1.5 class diagram. In addition to the implementation interfaces, there are provided and required interfaces. Interfaces can be represented in class diagrams as rectangles or as circles. For the sake of clarity of your diagrams, you can show or conceal interfaces.

Tip: Applying a provided interface link between a class and an interface creates a regular generalization/implementation link. To create provided interface, apply the provided interface link to a port on the client class.

UML 2.0 class diagram supports the ball-and socket notation for the provided and required interfaces. Choose Show as circle command on the context menu of the interface to obtain a lollipop between the client class and the supplier interface.

Sample Diagram

The figure below shows a class diagram with some of the new elements.



Class Diagram Types

There are two types of class diagrams used in Together:

- **Package (namespace)** diagrams. These diagrams are referred to as package diagrams in design projects, and namespace diagrams in implementation projects. They are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project group with the file extension `.txvpck`.
- **Logical** class diagrams. These diagrams are stored as XML files in the `ModelSupport_%PROJECTNAME%ModelSupport` folder of the project group with the file extension `.txvcls`.

Together automatically creates a default namespace diagram for the project and for each subdirectory under the project directory. The default project diagram is named default; the default namespace (package) diagrams are named after the respective namespaces (packages).

You create logical class diagrams manually by using the **Add ► Class Diagram** or **Add ► Other Diagram** command on the project context menu.

UML 2.0 Class Diagram Elements

The table below lists the elements of UML 2.0 class diagrams that are available using the **Tool Palette**.

UML 2.0 class diagram elements

Name	Type
Package	node
Class	node
Interface	node
Association Class	node
Port	node
Instance specification	node
Generalization/Implementation	link
Provided interface	link
Required interface	link
Association	link
Association end	link
Dependency	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Constraint	OCL node
Constraint link	OCL link
Note	annotation
Note Link	annotation link

Class Diagram Relationships

There are several kinds of relationships:

- **Association:** A relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes. Associations can be directed or undirected. A directed link points to the supplier class (the target). An association has two ends. An end may have a role name to clarify the nature of the association. A navigation arrow on an association shows which direction the association can be traversed or queried. A class can be queried about its Item, but not the other way around. The arrow also lets you know who "owns" the implementation of the association. Associations with no navigation arrows are bi-directional.
- **Generalization/Implementation:** An inheritance link indicating that a class implements an interface. An implementation has a triangle pointing to the interface.
- **Dependency**

There are several subtypes of an association relationship:

- **Simple Association**
- **Aggregation:** An association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole.
- **Composition**

Every class diagram has classes and associations. Navigability, roles, and multiplicities are optional items placed in a diagram to provide clarity.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers.

This table lists the most common multiplicities:

Multiplicities

Multiplicity	Meaning
0..1	Zero or one instance. The notation n . . m indicates n to m instances
0..* or *	No limit on the number of instances (including none)
1	Exactly one instance
1..*	At least one instance

Association Class and N-ary Association

Association classes appear in diagrams as three related elements:

- Association class itself (represented by a class icon)
- N-ary association class link (represented by a diamond)
- Association connector (represented by a link between both)

Association classes can connect to as many association end classes (participants) as required.

The **Object Inspector** of an association class, association link, and connector contain an additional Association tab. This tab contains the only label property, its value being synchronized with the name of the association class. For the association classes and association end links, the Custom node of the **Object Inspector** displays additional properties that corresponds to the role of this part of n-ary association ([associationClass](#) and [associationEnd](#) respectively).

You can delete each of the association end links or participant classes without destroying the entire n-ary association. However, deleting the association class results in deleting all the components of the n-ary association.

Inner Classifiers

The table below lists the diagram container elements along with the inner classifiers that you can add to container elements.

Inner classifiers

Container element	Inner classifiers available
Class	Class
	Interface
	Structure [C#]
	Delegate [C#]
	Delegate as Function
	Enum [C#]
Interface	Class]
	Interface
	Delegate
	Delegate as Function
	Enum
Structure	Class
	Interface
	Structure
	Delegate
	Delegate as
	Function
	Enum
Module	Class
	Interface
	Structure
	Delegate
	Delegate as
	Function
	Enum

Members

Note that the set of available members is different for the design and implementation projects.

The table below lists the diagram elements along with the members that can be added using the context menu of the element. The type of applicable project is specified in square brackets.

Members available

Container element	Members available
Class	Method [C#]
	Operation [Design]
	Constructor [Design, C#]
	Destructor [C#]
	Field [C#]
	Attribute [Design]
	Property [C#]
	Indexer [C#]
	Event [C#]
Interface	Method [C#]
	Property [C#]
	Indexer [C#]
	Event [C#]
	Attribute [Design]
	Operation [Design]
Structure	Method
	Constructor
	Field
	Property
	Indexer
	Event
Module	Function
	Subroutine
	Constructor
	Field
	Property
Enumeration	Enum Value

For implementation projects: If you set the abstract property for a method, property, or indexer (in abstract classes) as True in the Properties Window, the method body is removed from the source code. This is the desired behavior. Resetting the abstract property to False in the **Object Inspector**, adds a new empty method body.

UML 2.0 Use Case Diagrams

This section describes the elements of UML 2.0 Use Case Diagrams.

In This Section

[UML 2.0 Use Case Diagram Definition](#)

Provides UML 2.0 use case diagram definition.

[UML 2.0 Use Case Diagram Elements](#)

Describes UML 2.0 use case diagram elements.

[Extension Point](#)

Describes an extension point (Use Case diagrams).

UML 2.0 Use Case Diagram Definition

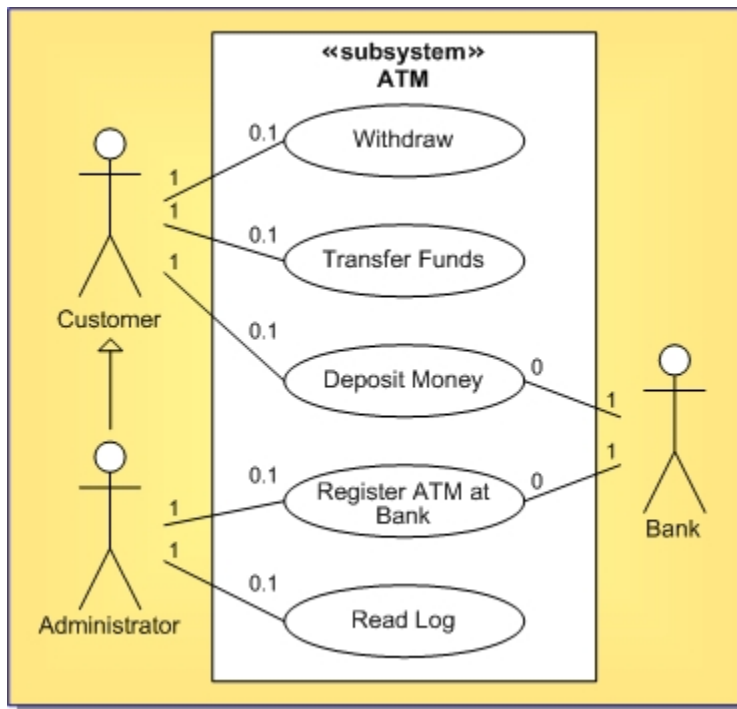
Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. p. 536.*

Definition

Use case diagrams describe required usages of a system, or what a system is supposed to do. The key concepts that take part in a use case diagram are actors, use cases, and subjects. A subject represents a system under consideration with which the actors and other subjects interact. The required behavior of the subject is described by the use cases.

Sample Diagram

The following diagram shows an example of actors and use cases for an ATM system.



UML 2.0 Use Case Diagram Elements

The table below lists the elements of UML 2.0 Use Case diagrams that are available using the **Tool Palette**.

UML 2.0 Use Case diagram elements

Name	Type	Description
Actor	node	Create extension points in the use cases in order to specify a point in the behavior of a use case, where this behavior can be extended by the behavior of some other use case. This element is available on the context menu of a use case.
Subject	node	
Use Case	node	
Extension point	node	
Extends	link	
Includes	link	
Generalization	link	
Association	link	
Node by Pattern	opens Pattern Wizard	
Link by Pattern	opens Pattern Wizard	
Note	annotation	
Note Link	annotation link	

Extension Point

An extension point refers to a location within a use case where you can insert action sequences from other use cases.

An extension point consists of a unique name within a use case and a description of the location within the behavior of the use case.

In a use case diagram, extension points are listed in the use case with the heading "Extension Points" (appears as bold text in the Diagram View).

UML 2.0 Interaction Diagrams

This section describes the elements of UML 2.0 Communication and Sequence diagrams.

In This Section

[UML 2.0 Sequence Diagram Definition](#)

Provides UML 2.0 sequence diagram definition and example.

[UML 2.0 Communication Diagram Definition](#)

Provides UML 2.0 communication diagram definition.

[UML 2.0 Sequence Diagram Elements](#)

Describes UML 2.0 sequence diagram elements.

[UML 2.0 Communication Diagram Elements](#)

Describes UML 2.0 communication diagram elements.

[Interaction](#)

Describes Interaction.

[UML 2.0 Message](#)

Describes UML 2.0 messages (Interaction diagrams).

[Execution Specification and Invocation Specification](#)

Describes an execution specification and invocation specification.

[Operator and Operand for a Combined Fragment](#)

About operator and operand for a combined fragment.

UML 2.0 Sequence Diagram Definition

Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication.

The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the message interchange between a number of lifelines. A Sequence Diagram describes an interaction by focusing on the sequence of messages that are exchanged.

UML 2.0 Communication Diagram Definition

Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication.

Communication Diagrams focus on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numbering scheme.

UML 2.0 Sequence Diagram Elements

The table below lists the elements of UML 2.0 sequence diagrams that are available using the **Tool Palette**.

UML 2.0 sequence diagram elements

Name	Type
Lifeline	node
Execution specification	node
Combined fragment	node
State invariant	node
Message	link
Interaction use	node
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

Sequence diagram can contain shortcuts to the other diagram elements. However, shortcuts to the elements that reside in the other interaction diagrams are not supported.

Interaction diagrams, represented in the **Model View**, display a number of auxiliary elements that are not visible in the **Diagram View**. These elements play supplementary role for representation of the diagram structure. Actually, these elements are editable, but it is strongly advised to leave them untouched, to preserve integrity of the interaction diagrams.

UML 2.0 Communication Diagram Elements

The table below lists the elements of UML 2.0 communication diagrams that are available using the **Tool Palette**.

UML 2.0 communication diagram elements

Name	Type
Lifeline	node
Message	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

Interaction diagrams, represented in the **Model View**, display a number of auxiliary elements that are not visible in the **Diagram View**. These elements play supplementary role for representation of the diagram structure. Actually, these elements are editable, but it is strongly advised to leave them untouched, to preserve integrity of the interaction diagrams.

Interaction

By using Together, you can create interactions for the detailed description and analysis of inter-process communications.

Interactions can be visually represented in your Together projects by means of the two most common interaction diagrams: Sequence and Communication. On the other hand, interactions can exist in projects without visual representation.

Interaction use

Within an interaction, you can refer to the other interactions described in your project. So called “Interaction use” elements serve this purpose. Note that referenced interaction can be explicitly defined from the model, or just specified as a text string.

Each interaction use is attached to its lifeline with a black dot. This dot is an individual diagram element. If an interaction use is expanded over several lifelines, you can delete the attachment dots from all lifelines but one. An interaction use should be connected with at least one lifeline.

Tie frame

Together makes it possible to spread combined fragments and interaction uses across several lifelines. This is done with the Tie Frame button of the diagram **Tool Palette**.

A frame can be attached to different points on the target lifeline. You choose the desired location on the target lifeline between the other elements and frames that belong to it. The frame shifts accordingly along the source lifeline.

It is important to understand that only those lifelines marked with dots are attached to the referenced interaction or combined fragment; lifelines that are only crossed by the frame are not attached. Attachment dots are separate diagram elements that can be selected and deleted.

Lifeline

A lifeline defines an individual participant of the interaction. A lifeline is shown in a sequence diagram as a rectangle followed by a vertical-dashed line.

Lifelines of an interaction can represent the parts defined in the class or composite structure diagrams. If the referenced element is multivalued, then the lifeline should have a selector that specifies which particular part is represented by this lifeline.

If a lifeline represents a connectable element, has type specified, or refers to another interaction, the Select menu becomes enabled on the context menu of this lifeline. Using this menu, you can navigate to the part, type or decomposition associated with the lifeline. These properties are defined by using the **Object Inspector**. If the represents property is set, the type and part properties are disabled.

You can define these properties manually by typing the values in the corresponding fields of the **Object Inspector**. If the specified values are not found in the model, they are displayed in single quotes. Such references are not related to any actual elements and the Select menu is not available for them. If the specified values can be resolved in the model, they are shown without quotes, and the Select menu is available for them.

State invariant

A state invariant is a constraint placed on a lifeline. This constraint is evaluated at runtime prior to execution of the next execution specification. State invariants are represented in the interaction diagrams in two ways: as OCL expressions or as references to the state diagrams. You can use the state invariants to provide comments to your interaction diagrams and to connect interactions with states.

It is important to note that Together provides validation of the state invariants represented as OCL expressions. If the syntax is wrong, or there is no valid context, the constraint is displayed red. For example, a lifeline should have type and represents properties defined to be a valid context.

UML 2.0 Message

Call messages are always visible in diagrams; reply messages normally are not displayed. However, you can visualize the reply message.

Messages on different diagram types

Messages in communication diagrams: When you draw a message between lifelines, a generic link line displays between the lifelines and a list of messages is created under it. The link line is present as long as there is at least one message between the lifelines.

Messages in sequence diagrams: Messages in sequence diagrams have the same properties as those in communication diagrams but allow you to perform more actions. The further discussion refers mainly to the sequence diagram messages.

Properties of the messages for both types of interaction diagrams can be edited in the **Object Inspector**.

Properties of the message links

Call messages have the following properties:

Property	Description
Sequence number	Use this field to view and edit the sequential number of a message. When the message number changes, the message call changes respectively.
Name	Displays the link name. This field can be edited.
Qualified name	A read-only field that displays the fully-qualified name of the message.
Stereotype	Use this field to define the message stereotype. The stereotype name displays above the link.
Signature	Use this field to specify the name of an operation or signal associated with the message. Note that changing the signature of a message call results in changing the signature of the corresponding reply.
Arguments	Displays actual arguments of an operation associated with a message call. This field can be edited.
Sort	Use this field to select the type of synchronization from the drop-down list. The possible values are:asynchCall, synchCall, asynchSignal. The message link changes its appearance accordingly. There are certain limitations related to the asynchronous calls: Sometimes it is impossible to create or paste an asynchronous call because of the frame limitations. Execution specification for an asynchronous call must always be located on a lifeline.
Show reply message	Use this Boolean option to define whether to draw a dashed return arrow.
Commentary	Use this textual field to enter comments for a message link.

Reply messages have the following properties:

Property	Description
Stereotype	Use this field to define the message stereotype.
Attribute	Use this field to define an attribute to which the return value of the message will be assigned. This field can be edited.
Signature	Use this field to specify the name of an operation or signal associated with the message. Note that changing the signature of a message reply results in changing the signature of the corresponding call.
Arguments	Displays arguments of an operation associated with a message call. This field can be edited. Note that changing the list of arguments of a reply message results in changing the corresponding call.

Return value	Displays the return value of an operation associated with a message link. This field can be edited.
Sort	Use this field to select the type of synchronization from the drop-down list. The possible values are:asynchCall, synchCall, asynchSignal. The message link changes its appearance accordingly.
Commentary	Use this text field to comment the link.

Note: Such properties of the call and reply messages as arguments, attribute, qualified name, return value, signature, and sort pertain to the invocation specification. You can edit these properties in the invocation specification itself, in the call or in the reply messages. As a result, the corresponding properties of the counterpart message and the invocation specification will change accordingly. Stereotype and commentary properties are unique for the call and reply messages.

Execution Specification and Invocation Specification

In sequence diagrams, Together automatically renders the execution specification of a message that shows the period of time when the message is active. When you draw a message link to the destination lifeline, the execution specification bar is created automatically. You can extend or reduce the period of time of a message by vertically dragging the top or bottom line of the execution specification as required.

It is also possible to create an execution specification on a lifeline without creating an incoming message link. In this case a found message is created, that is a message that comes from an object that is not shown in diagram. Use the **Object Inspector** to hide or show the found messages.

Messages in sequence diagrams have their origin in an invocation specification. This is an area within an execution specification. The notion of an invocation specification is introduced in Together's implementation of UML 2.0 sequence diagrams. Though this element is not defined in the UML 2.0 specification, it is a useful tool for modeling synchronous invocations with the reply messages. In particular, invocation specification marks a place where the reply messages (even if they are invisible) enter the execution context of a lifeline, and where sub-messages may reenter the lifeline.

Active and passive areas of the execution specification are rendered in different colors. The white execution specification bars denote active areas where you can create message links. The gray bars are passive and are not a valid source or target for the message links.

Operator and Operand for a Combined Fragment

About combined fragment

A combined fragment can consist of one or more interaction operators and one or more interaction operands. Number of interaction operands (just one, or more than one) depends on the last interaction operator of this combined fragment.

Use the **Tool Palette**, or context menus to create these elements. The operator type shows up in the descriptor in the upper-left corner of the design element. Note that you can define multiple operators in a combined fragment. In this case, the descriptor contains the list of all operators, which is a shorthand for nested operators.

When an operator is created, add the allowed operands, using the combined fragment's context menu.

A combined fragment can be expanded over several lifelines, detached from and reattached to lifelines. In the **Object Inspector**, use the Operators field to manage operators within the combined fragment.

Each combined fragment is attached to its lifeline with a black dot. This dot is an individual diagram element, which can be selected or deleted. Deleting a dot means detaching a combined fragment from the lifeline. Note that a combined fragment cannot be detached from all lifelines and should have at least one attachment dot.

You can reattach a combined fragment later, using the Tie Frame tool.

Operator

When a combined fragment is created, the operator is shown in a descriptor pentagon in the upper left corner of the frame. You can change the operator type, using the Operators field of the **Object Inspector**, which is immediately reflected in the descriptor.

The descriptor may contain several operators. UML 2.0 specification provides this notation for the nested combined fragments. In Together you can use this notation, or create nested combined fragment nodes.

Operand

Operands are represented as rectangular areas within a combined fragment, separated by the dashed lines. When a combined fragment is initially created, the number of operands is defined by the pattern defaults. Further, you can create additional operands, or remove the existing ones.

Note that the uppermost area of the operator is empty and does not contain any operands. It is reserved for the descriptor. Clicking on this area selects the entire operator; clicking on one of the dotted rectangles selects the corresponding operand. If a combined fragment contains only one operand, the entire combined fragment and the single existing operand are still separately selectable.

UML 2.0 State Machine Diagrams

This section describes the elements of UML 2.0 State Machine Diagrams.

In This Section

[UML 2.0 State Machine Diagram Definition](#)

Provides UML 2.0 state machine diagram definition and example.

[UML 2.0 State Machine Diagram Elements](#)

Describes UML 2.0 state machine diagram elements.

[State](#)

Describes a state (UML 1.5 Activity, UML 1.5 Statechart, UML 2.0 State Machine diagrams).

[Transition](#)

Describes a transition (UML 1.5 Activity, UML 1.5 Statechart, UML 2.0 State Machine diagrams).

[Deferred Event](#)

Describes a deferred event.

[History Element \(State Machine Diagrams\)](#)

Describes UML 2.0 history.

UML 2.0 State Machine Diagram Definition

States are the basic units of the state machines. In UML 2.0 states can have substates.

Execution of the diagram begins with the Initial node and finishes with Final or Terminate node or nodes. Please refer to UML 2.0 Specification for more information about these elements.

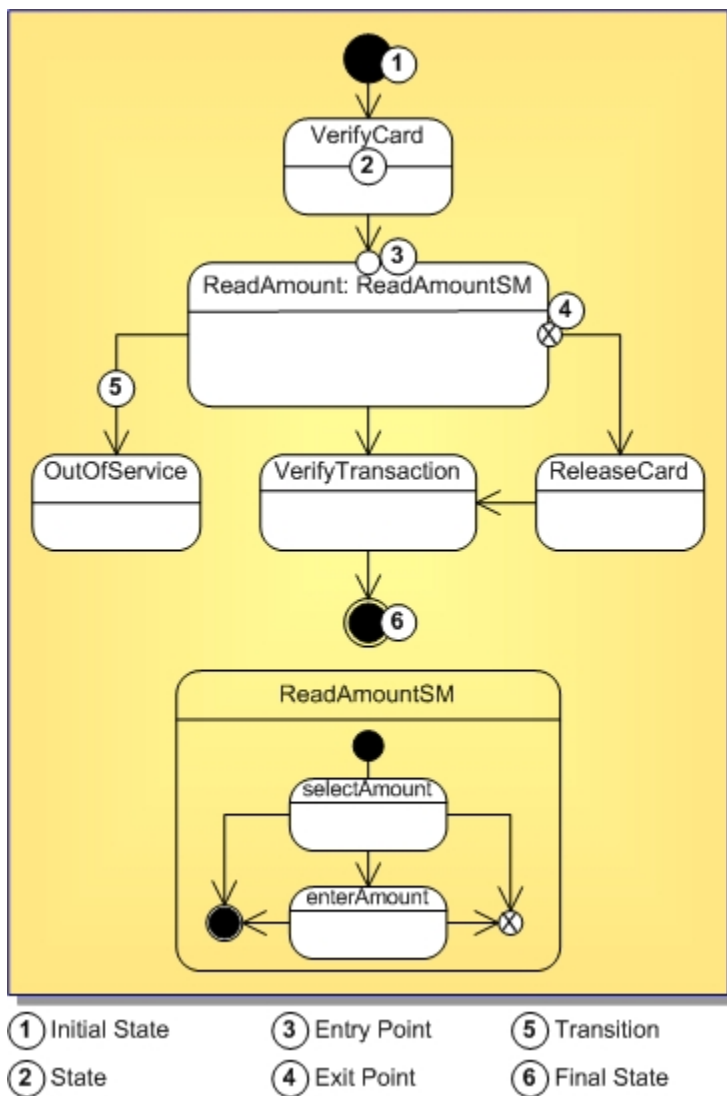
Definition

State Machine diagrams describe the logic behavior of the system, a part of the system, or the usage protocol of it.

On these diagrams you show the possible states of the objects and the transitions that cause a change in state.

State Machine diagrams in UML 2.0 are different in many aspects compared to Statechart diagrams in UML 1.5.

Sample Diagram



UML 2.0 State Machine Diagram Elements

The table below lists the elements of UML 2.0 State Machine diagrams that are available using the **Tool Palette**.

UML 2.0 State Machine diagram elements

Name	Type	Description
State	node	
Entry point	node	Execution of the state starts at this point. It is possible to create several entry points for one state, that makes sense if there are substates.
Exit point	node	Execution of the state finishes at this point. It is possible to create several exit points for one state, that makes sense if there are substates.
Initial	node	
Final	node	
Terminate	node	
Shallow history	node	
Deep history	node	
Region	node	Use regions inside the states to group the substates. The regions may have different visibility settings and history elements. Each state has one region immediately after creation (though it can be deleted.) In the regions, you can create all the elements that are available for the State Machine diagram. only available on the state context menu
Fork	node	
Join	node	
Choice	node	
Junction	node	
Transition	link	Draw a link from the exit point of source state (or the state without exit points) to the entry point of the destination (or the state without points).
Internal transition	link	Internal transition elements are only available on the state context menu.
Node by Pattern	opens Pattern Wizard	
Link by Pattern	opens Pattern Wizard	
Note	annotation	
Note Link	annotation link	

State

A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (for example, the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).

Actions

Entry and exit actions are executed when entering or leaving a state, respectively.

You can create these actions in statechart diagrams as special nodes, or as stereotyped internal transitions.

Composite (nested) state

Create a composite state by nesting one or more levels of states within one state. You can also place start/end states and a history state inside of a state, and draw transitions among the contained substates.

Transition

A single transition comes out of each state or activity, connecting it to the next state or activity.

A transition takes operation from one state to another and represents the response to a particular event. You can connect states with transitions and create internal transitions within states.

Internal transition

An internal transition is a way to handle events without leaving a state (or activity) and dispatching its exit or entry actions. You can add an internal transition to a state or activity element.

An internal transition is shorthand for handling events without leaving a state and dispatching its exit or entry actions.

Self-transition

A self-transition flow leaves the state (or activity) dispatching any exit action(s), then reenters the state dispatching any entry action(s). You can draw self-transitions for both activity and state elements on an activity diagram.

Self-transition for statechart diagrams

Self-transition for activity diagrams

Multiple transition

A transition can branch into two or more mutually-exclusive transitions.

A transition may fork into two or more parallel activities. A solid bar indicates a fork and the subsequent join of the threads coming out of the fork.

A transition may have multiple sources (a join from several concurrent states) or it may have multiple targets (a fork to several concurrent states).

You can show multiple transitions with either a vertical or horizontal orientation in your State and Activity diagrams. Both the State and Activity diagram toolbars provide separate horizontal and vertical fork/join buttons for each orientation. The two orientations are semantically identical.

Guard expressions

All transitions, including internal ones, are provided with the guard conditions (logical expressions) that define whether this transition should be performed. Also you can associate a transition with an effect, which is an optional activity performed when the transition fires. The guard condition is enclosed in the brackets (for example, "[false]") and displayed near the transition link on a diagram. Effect activity is displayed next to the guard condition. You can define the guard condition and effect using the **Object Inspector**.

Guard expressions (inside []) label the transitions coming out of a branch. The hollow diamond indicates a branch and its subsequent merge that indicates the end of the branch.

Deferred Event

A deferred event is like an internal transition that handles the event and places it in an internal queue until it is used or discarded.

A deferred event may be thought of as an internal transition that handles the event and places it in an internal queue until it is used or discarded. You can add a deferred event to a state or activity element.

History Element (State Machine Diagrams)

The Shallow History and Deep History elements are placed on regions of the states.

There may be none or one Deep History, and none or one Shallow History elements in each region. If there is only one history element in a region, it may be switched from the Deep to Shallow type by changing its kind property.

Please refer to UML 2.0 Specification for more information about these elements.

UML 2.0 Activity Diagrams

This section describes the elements of UML 2.0 Activity Diagrams.

In This Section

[UML 2.0 Activity Diagram Definition](#)

Provides UML 2.0 activity diagram definition.

[UML 2.0 Activity Diagram Elements](#)

Describes UML 2.0 activity diagram elements.

[Pin](#)

Describes pins (UML 2.0 Activity Diagrams).

UML 2.0 Activity Diagram Definition

Definition

The activity diagram enables you to model the system behavior, including the sequence and conditions of execution of the actions. Actions are the basic units of the system behavior.

An Activity diagram enables you to group and ungroup actions. If an action can be broken into a sequence of other actions, you can create an activity to represent them.

In UML 2.0, activities consist of actions. Actions are not states (compared to UML 1.x) and can have subactions. An action represents a single step within an activity, that is, one that is not further decomposed within the activity. An activity represents a behavior which is composed of individual elements that are actions. An action is an executable activity node that is the fundamental unit of executable functionality in an activity, as opposed to control and data flow among actions. The execution of an action represents some transformation or processing in the modeled system, be it a computer system or otherwise.

The semantics of activities is based on token flow. By flow, we mean that the execution of one node affects and is affected by the execution of other nodes, and such dependencies are represented by edges in the activity diagram. Data and control flows are different in UML 2.0.

A control flow may have multiple sources (it joins several concurrent actions) or it may have multiple targets (it forks into several concurrent actions).

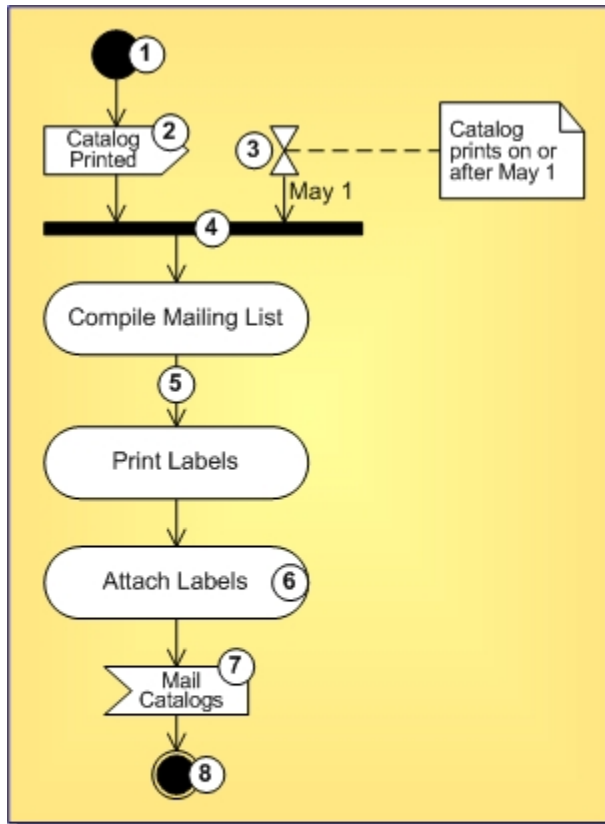
Each flow within an activity can have its own termination, which is denoted by a flow final node. The flow final node means that a certain flow within an activity is complete. Note that the flow final may not have any outgoing links.

Using decisions and merges, you can manage multiple outgoing and incoming control flows.

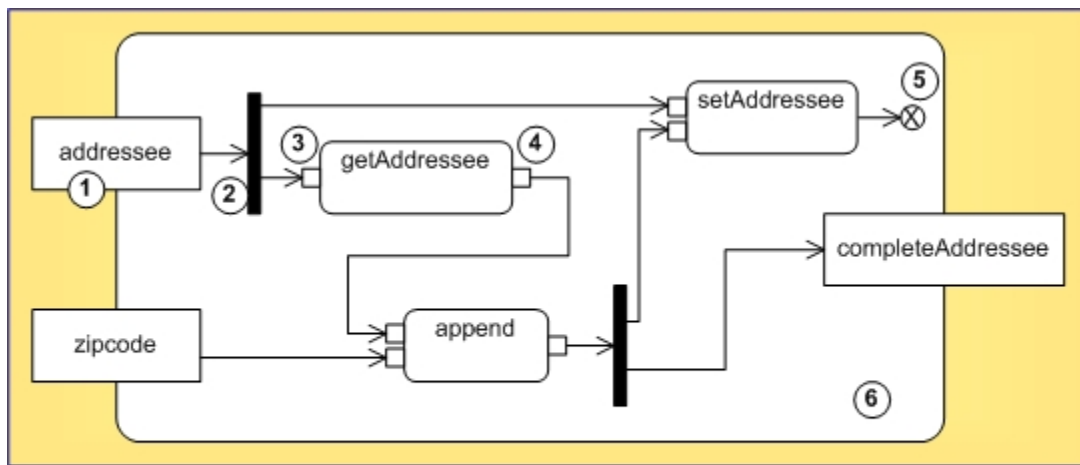
Use the **Object Inspector** to adjust action properties, including:

- In the Properties, View, Description, and Custom tabs, configure standard properties of the element.
- In the Local Precondition and Local Postcondition tabs, select the language of the constraint expression from the Language list box. The possible options are OCL and plain text. In the edit field below the list box, enter the constraint expression for this action.

Sample Diagram



- | | |
|----------------------------|-----------------------|
| ① Initial State | ⑤ Control Flow |
| ② Send Signal Action | ⑥ Action |
| ③ Accept Time Event Action | ⑦ Accept Event Action |
| ④ Join | ⑧ Final State |



- | | | |
|----------------------|--------------|--------------|
| ① Activity Parameter | ③ Input Pin | ⑤ Flow Final |
| ② Fork | ④ Output Pin | ⑥ Activity |

UML 2.0 Activity Diagram Elements

The table below lists the elements of UML 2.0 Activity diagrams that are available using the **Tool Palette**.

UML 2.0 Activity diagram elements

Name	Type
Activity	node
Activity parameter	node component
Action	node
Initial	node
Activity final	node
Decision	node
Merge	node
Flow Final	node
Control Flow	link
Input pin	pin
Output pin	pin
Value pin	pin
Object node	node
Central Buffer	node
Data Store	node
Object Flow	link
Accept Event Action	node
Accept Time Event Action	node
Send Signal Action	node
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

Pin

Actions can consume some input values or produce some output values. Input, Output and Value pins hold such values.

UML 2.0 Component Diagrams

This section describes the elements of UML 2.0 Component diagrams.

In This Section

[UML 2.0 Component Diagram Definition](#)

Provides UML 2.0 component diagram definition.

[UML 2.0 Component Diagram Elements](#)

Describes UML 2.0 component diagram elements.

UML 2.0 Component Diagram Definition

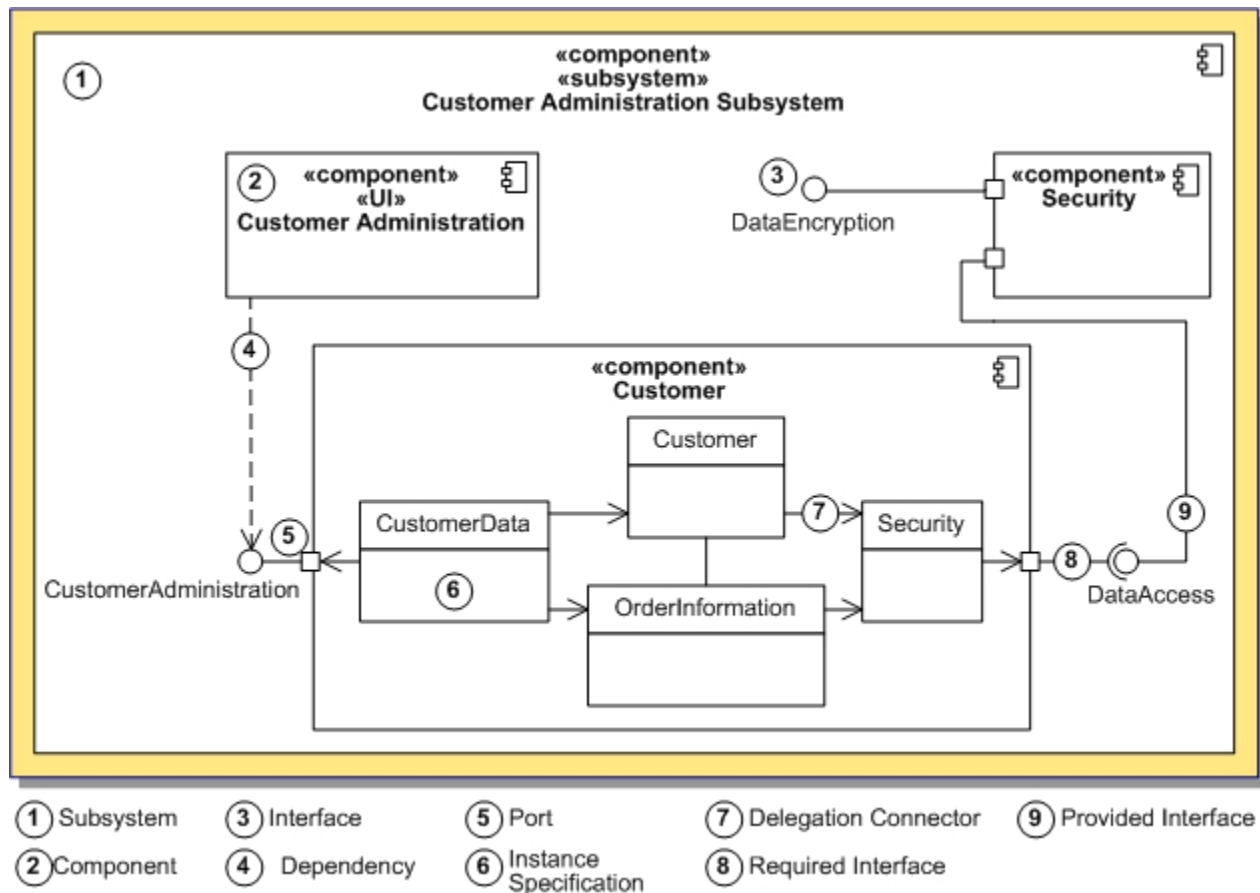
This topic describes the UML 2.0 Component Diagram.

Definition

According to the UML 2.0 specification, a component diagram can contain instance specifications. An instance specification can be defined by one or more classifiers. You can use classes, interfaces, or components as a classifiers. You can instantiate a classifier using the **Object Inspector**, or the in-place editor.

Sample Diagram

The following component diagram specifies a set of constructs that can be used to define software systems of arbitrary size and complexity.



UML 2.0 Component Diagram Elements

The table below lists the elements of UML 2.0 component diagrams that are available using the **Tool Palette**.

UML 2.0 component diagram elements

Name	Type
Component	node
Class	node
Port	node
Artifact	node
Interface	node
Instance specification	node
Delegation connector	link
Provided interface	link
Required interface	link
Association	link
Aggregation	link
Dependency	link
Realization	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

UML 2.0 Deployment Diagrams

This section describes the elements of UML 2.0 Deployment diagrams.

In This Section

[UML 2.0 Deployment Diagram Definition](#)

Provides UML 2.0 deployment diagram definition.

[UML 2.0 Deployment Diagram Elements](#)

Describes UML 2.0 deployment diagram elements.

[Class Diagram Relationships](#)

Describes class diagram relationships.

UML 2.0 Deployment Diagram Definition

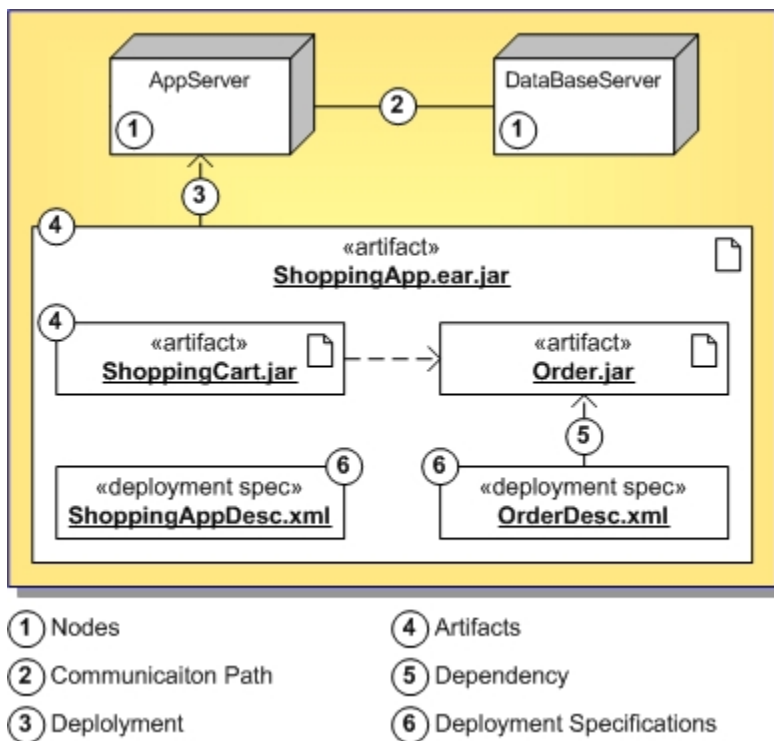
This topic describes the UML 2.0 Deployment Diagram.

Definition

The deployment diagram specifies a set of constructs that can be used to define the execution architecture of systems that represent the assignment of software artifacts to nodes. Nodes are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. Artifacts represent concrete elements in the physical world that are the result of a development process.

Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. pp. 207, 212.*

Sample Diagram



UML 2.0 Deployment Diagram Elements

The table below lists the elements of UML 2.0 deployment diagrams that are available using the **Tool Palette**.

UML 2.0 deployment diagram elements

Name	Type
Node	node
Artifact	node
Device	node
Execution specification	node
Deployment specification	node
Instance specification	node
Deployment	link
Generalization	link
Association	link
Dependency	link
Manifestation	link
Communication path	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

An artifact represents a physical entity and is depicted in diagram as a rectangle with the <<artifact>> stereotype. An artifact may have properties which define its features, and operations which can be performed on its instances. Physically the artifacts can be model files, source files, scripts, binary executable files, a table in a database system, a development deliverable, a word-processing document, or a mail message. A deployed artifact is one that has been deployed to a node used as a deployment target. Deployed artifacts are connected with the target node by deployment links.

Artifacts can include operations.

You can create complex artifacts, by nesting artifact icons.

UML 2.0 Composite Structure Diagrams

This section describes the elements of UML 2.0 Composite Structure Diagrams.

In This Section

[UML 2.0 Composite Structure Diagram Definition](#)

Provides UML 2.0 composite structure diagram definition.

[UML 2.0 Composite Structure Diagram Elements](#)

Describes UML 2.0 composite structure diagram elements.

[Delegation Connector](#)

Describes a delegation connectors.

UML 2.0 Composite Structure Diagram Definition

Diagram courtesy of the Unified Modeling Language: *Superstructure version 2.0. August 2003. p. 178.*

Definition

Composite structure diagrams depict the internal structure of a classifier, including its interaction points to the other parts of the system. It shows the configuration of parts that jointly perform the behavior of the containing classifier.

A collaboration describes a structure of collaborating parts (roles). A collaboration is attached to an operation or a classifier through a Collaboration Use.

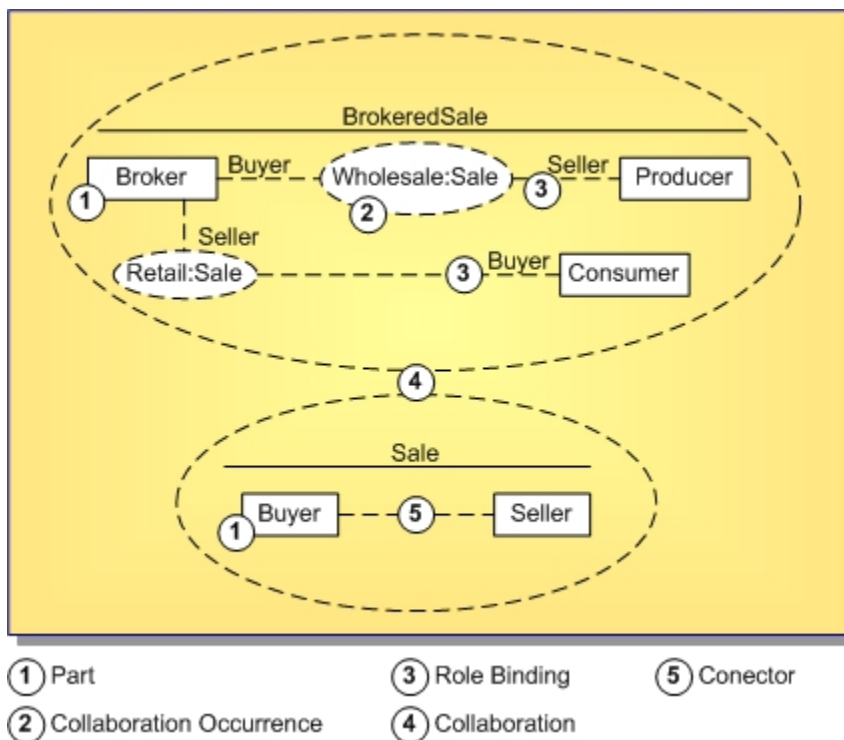
Classes and collaborations in the Composite Structure diagram can have internal structure and ports. Internal structure is represented by a set of interconnected parts (roles) within the containing class or collaboration. Participants of a collaboration or a class are linked by the connectors.

A port can appear either on a contained part, or on the boundary of the class.

The contained parts can be included by reference. Referenced parts are represented by the dotted rectangles.

Composite Structure diagram supports the ball-and-socket notation for the provided and required interfaces. Interfaces can be shown or hidden in the diagram as needed.

Sample Diagram



UML 2.0 Composite Structure Diagram Elements

The table below lists the elements of UML 2.0 composite structure diagrams that are available using the **Tool Palette**.

UML 2.0 composite structure diagram elements

Name	Type
Class	node
Interface	node
Collaboration	node
Collaboration Occurrence	node
Part	node
Referenced part	node
Port	node
Provided interface	link
Required interface	link
Connector	link
Collaboration role	link
Role binding	link
Node by Pattern	opens Pattern Wizard
Link by Pattern	opens Pattern Wizard
Note	annotation
Note Link	annotation link

Delegation Connector

An interface can delegate its obligations to another interface through the delegation connector.

Together Refactoring Operations

The following refactoring operations are available in Together:

Refactoring operations

Operation	Description
Change parameters	You can rename, add, or remove parameters for a single method using the Model View , Diagram View , or the Editor.
Extract interface	The Extract Interface command creates a new interface from one or more selected classes. Each selected class should implement the interface.
Extract method	You can extract method from a class or interface using the Editor only. It is important that the code fragment to be extracted must contain complete statements.
Extract superclass	The Extract Superclass command creates an ancestor class from several members of a given class.
Inline variable	If you have a temporary variable that is assigned once with a simple expression, you can replace all references to that variable with the expression using the Inline Variable command in the Developer Studio 2006 Editor.
Introduce field	Creates a new field.
Introduce variable	Creates a new variable. This command is available in the Editor.
Move members	Moving members only applies to the static methods, static fields and static properties (static members). This command is available on the Diagram View, on the Model View , and in the Editor.
Pull Members Up	Use this command to copy a member (field, method, property, indexer, and event) from a subclass to a superclass, optionally making it abstract. If there are no superclasses, an error message is displayed.
Push Members Down	Use this command to copy a member (field, method, property, and event) from a superclass to a subclass, optionally making it abstract. If there are no subclasses, an error message is displayed. Indexers cannot be pushed down.
Safe Delete	The Safe Delete command searches your code for any usages of the element that you wish to delete. You can invoke the command from the Diagram View , Model View , or from the Editor.

Project Types and Formats with Support for Modeling

There are two basic project types:

- **Design project.** Project file extension: `.bdsproj` `.tgproj`. These projects are language-neutral and comply with one of the two versions of UML specifications: UML 1.5 or UML 2.0.
- **Implementation project.** Project file extension: `.bdsproj`. You can create models for language-specific projects. Modeling that complies with UML 1.5 specification is supported for C# and Delphi projects. Together modeling features are automatically activated for these projects.

There are multiple project formats of the types mentioned above, supported by Together:

Supported project formats

Project format	Project type	Basic supported actions
Delphi formats	Implementation	Create, open, save, edit
C# .NET	Implementation	Create, open, save, edit
Together design formats: UML 1.5, UML 2.0	Design	Create, open, save, edit
Other editions of Together	Design or implementation	Import, share
IBM Rational Rose (MDL) format	Design	Create a new design project by using the import wizard
XMI format	Design	Import, export

Index

- @ operator
 - operators, 74
- algorithm
 - hierarchical, 711
 - orthogonal, 713
 - spring embedder, 713
 - Together, 712
 - tree, 712
- arithmetic operators
 - operators, 70
- array parameters
 - parameters, 153
- array properties
 - properties, 187
- arrays, 100
- assembler syntax
 - inline assembler, 287
- assembly expression
 - inline assembler, 293
- assembly procedures and functions
 - inline assembler, 303
- assignment statements
 - statements, 52
- automation objects
 - interfaces, 255
- binding
 - methods, 174
- blocks, 66
- blocks and scope, 66
- boolean operators
 - operators, 70
- boolean types
 - data types, 87
- calling conventions
 - procedures and functions, 142
- calling procedures and functions
 - procedures and functions, 159
- case statements, 58
- character set (Delphi)
 - reserved words, 45
 - Delphi character set, 45
- character types
 - data types, 87
- circular unit references
 - uses clause, 36
- class constructor
 - methods, 180
- classes and objects, 165
- class fields
 - fields, 172
- class helpers
 - classes and objects, 215
- class methods
 - methods, 177
- class properties
 - properties, 191
- class references
 - classes and objects, 197
- class static methods
 - methods, 178
- class types
 - classes and objects, 165
- Close function
 - device drivers, 223
- combined fragment, 849
- comments (Delphi)
 - compiler directives (Delphi), 49
- compound statements, 55
- constants
 - data types, 131
- constructors
 - methods, 179
 - Delphi for .NET, 273
- custom attributes, 305
- data types, variables, and constants, 83
- declarations, 51
- declarations and statements, 51
- declaring exception types
 - exceptions, 202
- declaring types
 - data types, 127
- default parameters
 - parameters, 155
- delegation, implementing interfaces by
 - interfaces, 251
- Delphi language overview, 23
- destructors
 - methods, 181
- device drivers, 221
- diagram layout options
 - general group, 711
- Directives (Delphi), 48
- dispatch interface methods
 - automation objects, 255
- Dispose
 - Delphi for .NET, 274
- DLLs, calling, 231
- dynamically allocated multidimensional arrays
 - arrays, 104
- dynamically loaded libraries

- libraries, 233
- dynamic arrays
 - arrays, 101
- enumerated types
 - data types, 88
- event properties and event handlers
 - events, 193
- events
 - classes and objects, 193
- exceptions
 - classes and objects, 201
- export clauses
 - libraries, 233
- expressions, 69
- external declarations
 - procedures and functions, 143
- fields
 - classes and objects, 171
- file input and output
 - standard routines, 219
- file types, 109
- filters
 - member, 719
 - node, 720
- finalization section, 34
- Finalize
 - Delphi for .NET, 273
- Flush function
 - device drivers, 222
- for...in statements, 62
- for loops, 61
- forward and interface declarations
 - procedures and functions, 143
- forward declarations
 - classes and objects, 170
- function calls
 - expressions, 76
- function declarations
 - procedures and functions, 140
- goto statements, 53
- guard expression, 779
- hinting directives
 - compiler directives, delphi, 51
- identifiers (Delphi)
 - character set (Delphi), 46
- if statements
 - with statements, 54
- Interface, 246
- implementation section, 33
- implementing interfaces
 - interfaces, 249
- importing functions
 - procedures and functions, 144
- indexes
 - expressions, 77
- index specifiers
 - properties, 188
- inheritance and scope
 - classes and objects, 166
- inherited methods
 - methods, 174
- initialization section, 34
- inline assembler, 285
- inline directive, 160
- InOut function
 - device drivers, 222
- integer types
 - data types, 85
- interaction use, 843
- interface, 811
- interface references
 - interfaces, 253
- Interfaces
 - GUID, 246
- interface section, 33
- labels (Delphi), 48
- libraries
 - initialization code, 235
- libraries and packages, 231
- linking to object files
 - procedures and functions, 143
- local declarations
 - procedures and functions, 147
- logical (bitwise) operators
 - operators, 71
- long strings
 - string types, 94
- loops, 60
- memory management
 - Delphi for Win32, 261
 - Delphi for .NET, 273
- message methods
 - methods, 182
- methods
 - classes and objects, 173
- multicast events
 - events, 195
- multidimensional dynamic arrays
 - arrays, 102
- namespaces

- programs and units, 39
- namespaces, declaring, 39
- namespaces, multi-unit, 42
- namespaces, searching, 40
- Nested Constants
 - Nested Types, 210
- nested exceptions
 - exceptions, 206
- nested types
 - classes and objects, 209
- null-terminated string functions
 - strings, 223
- null-terminated strings
 - strings, 223
- numerals (Delphi)
 - character set (Delphi), 48
- object interfaces
 - interfaces, 245
- open array parameters
 - parameters, 153
- open arrays, 159
- Open function
 - device drivers, 222
- operand, 849
- operator, 849
- operator overloading
 - classes and objects, 211
- operators
 - expressions, 69
 - classes and objects, 198
- options
 - general, 705
 - generate documentation general, 721
 - generate documentation include, 721
 - generate documentation navigation, 722
 - Together Diagram Appearance General, 707
 - Together Diagram Appearance Grid, 708
 - Together Diagram Appearance Nodes, 708
 - Together Diagram Appearance UML In Color, 709
 - Together support, 705
- ordinal types
 - data types, 85
- overloading methods
 - methods, 178
- packages, 239
 - compiling, 241
- parameters
 - procedures and functions, 149 159
- parameters, passing, 279
- pointer operators
 - operators, 72
- pointer types
 - data types, 111
- precedence of operators
 - operators, 75
- procedural types
 - data types, 115
- procedure declarations
 - procedures and functions, 139
- procedures and functions, 139
 - overloading, 144
- program control, 279
- program heading, 31
- program organization
 - programs and units, 31
- program structure, 31
- properties
 - classes and objects, 185
- property access, 185
- property overrides
 - properties, 189
- raising and handling
 - exceptions, 202
- re-raising exceptions
 - exceptions, 205
- real types
 - data types, 91
- records (advanced)
 - records, 108
- records (traditional)
 - records, 105
- relational operators
 - operators, 73
- repeat loops, 60
- reserved words (Delphi), 47
- scope, 67
- Self identifier
 - methods, 174
- set constructors
 - expressions, 76
- set operators
 - operators, 73
- sets, 99
- shortcuts
 - other, 696
- short strings
 - string types, 94
- simple types
 - data types, 85
- standard exception classes and routines
 - exceptions, 207

- standard routines
 - System unit, 219
- state invariant, 843
- statements, 52
 - simple statements, 52
- static arrays
 - arrays, 100
- static methods
 - methods, 175
- storage specifiers
 - properties, 189
- strict visibility
 - classes and objects, 168
- string operators
 - operators, 72
- string parameters
 - parameters, 152
- strings, 48
- string types
 - data types, 93
- structured types
 - data types, 99
- subrange types
 - data types, 90
- symbols (Delphi)
 - character set (Delphi), 45
- syntax (Delphi)
 - character set (Delphi), 45
- Text file type
 - file types, 220
- tie frame, 843
- TObject and TClass
 - classes and objects, 166
- transition
 - multiple, 779 779 779
- try...except statements
 - exceptions, 203
- try... finally statements
 - exceptions, 206
- typecasts
 - expressions, 77
- type compatibility
 - data types, 123
- unit names, 41
- units
 - Delphi for .NET, 275
- unit structure, 32
- untyped files
 - file types, 221
- uses clause, 32
 - unit references, 34
- variables
 - data types, 129
- variant parts in records, 106
- variant types
 - data types, 119
- virtual and dynamic methods
 - methods, 175
- visibility
 - classes and objects, 168
- when to use exceptions
 - exceptions, 201
- while loops, 60
- wide-character strings, 224
- WideString
 - string types, 95
- with statements, 55