

Welcome to RoboHELP. Click Topic (Ctrl+T) to add your first Help topic.

Dynamic-Link Libraries (DLLs)

[See also](#)

A dynamic-link library (DLL) is an executable module (extension .DLL) that contains code or resources that are used by other DLLs or applications. In the Windows environment, DLLs permit multiple applications to share code and resources.

The C++Builder concept most comparable to a DLL is a unit. However, routines in units are linked into your executable file at link time (statically linked), whereas DLL routines reside in a separate file and are made available at run time (dynamically linked).

DLLs provide the ability for multiple applications to share a single copy of a routine they have in common. The .DLL file must be in the same directory as the application at run time.

When the program is loaded into memory, the application dynamically links the **procedure** and **function** calls in the program to their entry points in the DLLs used by the program.

Note: DLLs can export procedures and functions only.

C++Builder applications can use DLLs that were not written in Object Pascal. Also, programs written in other languages can use DLLs written in Object Pascal.

For more information on using DLLs, choose from the following topics:

[Accessing routines stored in DLLs](#)

[Writing DLLs](#)

See also

[Function calls](#)

[Import units](#)

Accessing routines stored in DLLs

[See also](#) [Example](#)

There are two ways to access and call a routine stored in a DLL.

- Use an external declaration in your program (static import or implicit loading).
Using an **external** declaration to perform a static DLL import causes the DLL to be loaded before execution of your program begins. In this case you cannot change the name of the DLL at run time. Your program cannot be executed if it specifies a DLL that isn't available at run time.
- Use GetProcAddress and LoadLibrary WinAPI calls to initialize procedure pointers in your program (dynamic import or explicit loading).

Using GetProcAddress and LoadLibrary (the two must be used in conjunction) to import a DLL gives your program control over what DLL file is actually loaded. For example, Windows device drivers are all DLLs with the same interface, but that internally perform hardware-specific functions. Programs can use the device driver DLLs without knowing anything about the hardware. With a dynamic import, even if LoadLibrary fails to locate a DLL your program can continue to run.

Although a DLL can have variables, it is not possible to import them into other modules. Any access to a DLL's variables must take place through a procedural interface.

When you compile a program that uses a DLL, the compiler does not look for the DLL, so it need not be present.

If you write your own DLLs, you must compile them separately.

Importing routines

In imported procedures and functions, the **external** directive takes the place of the declaration and statement parts that would otherwise be present.

Object Pascal provides three ways to import procedures and functions:

- by name
- by new name
- by ordinal number

By name

When you import a routine from a DLL with no index or name clause specified, the procedure or function is imported explicitly by name.

The name used is the procedure's or function's identifier, with the same spelling and case.

When a name clause is specified, the procedure or function is imported by a different name than its identifier.

Note: The DLL name specified after the **external** keyword and the new name specified in a name clause do not have to be string literals. Any constant string expression is allowed.

By new name

When you import a routine from a DLL with a name clause specified, the procedure or function is imported by a different name than its identifier.

By ordinal number

When you import a routine from a DLL with an index clause present, the procedure or function is imported by ordinal.

Importing by ordinal reduces the load time of the module because Windows does not have to look up the name in the DLL's name table.

The ordinal number specified in an index clause can be any constant-integer expression.

See also

[Import units](#)

[Writing DLLs](#)

Example

Example for importing routines by name

Example for importing routines by new name

Example for importing routines by ordinal number

Example

{ The following example imports the ImportByName procedure from testlib.dll using the name 'ImportByName'. }

```
procedure ImportByName; external 'testlib.dll';
```

Example

{ The following example imports the ImportByNewName procedure from testlib.dll using the name 'RealName'. }

```
procedure ImportByNewName; external 'testlib.dll' name 'RealName';
```


Example

{ The following example imports the ImportByOrdinal procedure as the fifth entry point in testlib.dll. }

```
procedure ImportByOrdinal; external 'testlib.dll' index 5;
```

Import units

You can place declarations of imported procedures and functions directly in the program that imports them. Usually, though, they are grouped together in an import unit that contains declarations for all procedures and functions in a DLL, along with any constants and types required to interface with the DLL.

To use an import unit,

- ▶ Add it to the uses clause of the calling unit.

Import units are not a requirement of the DLL interface, but they do simplify maintenance of projects that use multiple DLLs. Also, when the associated DLL is modified, only the import unit needs updating to reflect the changes.

When you compile a program that uses a DLL, the compiler does not look for the DLL so it need not be present. However, when you run the program it must be present.

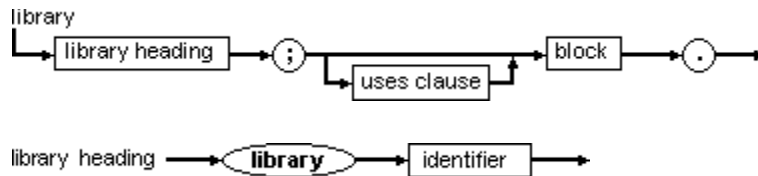
If you write your own DLLs, you must compile them separately.

Writing DLLs

See also [Example](#)

The structure of an Object Pascal DLL is identical to that of a program, except that a DLL starts with a library header instead of a **program** header.

The **library** header tells C++Builder to produce an executable file with the extension .DLL instead of .EXE. It also ensures that the executable file is marked as being a DLL.



If procedures and functions are to be exported by a DLL, they will often be compiled using the **stdcall** procedure directive. Although not a requirement, use of the **stdcall** calling convention makes it possible for applications written in other languages to use the DLL.

To actually export the routines, use the exports clause.

Libraries often consist of several units. In such cases, the library source file itself is frequently reduced to a uses clause, an **exports** clause, and the library's initialization code.

Global variables

Global variables declared in a DLL are private to that DLL.

A DLL cannot access variables declared by modules that call the DLL, and it is not possible for a DLL to export its variables for use by other modules. Such access must take place through a procedural interface.

Even though a DLL can be used by multiple applications at the same time, to the DLL it appears that there is only one client, and each instance of the DLL will have its own set of global variables. For multiple DLLs (or multiple instances of one DLL) to share memory, the DLLs must use memory mapped files. See the Windows API documentation for further details on this topic.

Example

{ The following example implements a very simple DLL with two exported functions: }

library MinMax;

{ The **stdcall** procedure directive exports Min and Max with a calling convention supported by other languages. }

```
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;
```

{The **exports** clause actually exports the two routines, supplying an optional ordinal number for each of them}

exports

```
  Min index 1,  
  Max index 2;
```

```
begin  
end.
```

See also

[Accessing routines stored in DLLs](#)

[DLLs and the system unit](#)

[Import units](#)

[Library initialization code](#)

[Run-time errors in DLLs](#)

[The Shared Memory Manager](#)

[The DLLProc variable](#)

Library initialization code

[Example](#) [Writing DLLs](#)

The statement part of a library constitutes the library's initialization code. The initialization code is executed once, when the library is loaded.

A library's initialization code typically performs tasks like registering window classes for window procedures contained in the library, and setting initial values for the library's global variables. In addition, the initialization code of a library can install an exit procedure using the ExitProc variable. The exit procedure will be executed when the operating system unloads the library.

The initialization code of a library can signal an error condition by setting the ExitCode variable to a non-zero value. ExitCode is declared in the System unit and defaults to zero, indicating successful initialization. If the initialization code sets ExitCode to a non-zero value, the DLL is unloaded from memory and the calling application is notified of the failure to load the DLL.

Note If an unhandled exception occurs while executing a library's initialization code, the calling application will be notified of a failure to load the DLL.

When a DLL is unloaded, C++Builder executes the library's exit procedures by continuing to call the address stored in the ExitProc variable until ExitProc becomes **nil**. Because this works the same way as exit procedures are handled in Object Pascal programs, you can use the same exit procedure logic in both programs and libraries.

Note The initialization parts of all units used by an application or library are always executed before the application or library's statement part. Likewise, unit finalization parts are executed after an exit procedure installed by an application or library (unit finalization parts in fact use the ExitProc variable to install themselves).

Example

{The following code is an example of a library with initialization code and an exit procedure:}

```
library Test;
```

```
var
```

```
    SaveExit: Pointer;
```

```
procedure LibExit;
```

```
begin
```

```
    :
```

```
    { Library exit code }
```

```
    :
```

```
    ExitProc := SaveExit;          { Restore exit procedure chain }
```

```
end;
```

```
begin
```

```
    :
```

```
    { Library initialization code }
```

```
    :
```

```
    SaveExit := ExitProc;          { Save exit procedure chain }
```

```
    ExitProc := @LibExit;          { Install LibExit exit procedure }
```

```
end.
```

DLLs and the system unit

[See also](#)

The IsLibrary Boolean variable can be used to determine whether code is executing in the context of an application or a library. IsLibrary is always False in an application, and always True in a library.

During a DLL's lifetime, the HInstance variable contains the instance handle of the DLL.

The CmdLine variable is always **nil** in a DLL.

See also
[Writing DLLs](#)

Exceptions and run-time errors in DLLs

[See also](#)

If an exception is raised but not handled in a DLL, the exception is propagated out of the DLL. If the calling application or DLL is itself written in C++Builder, it is possible to handle the exception through a normal **try...except** statement.

If the calling application or DLL is written in another programming language, the exception can be handled as an operating system exception with an exception code of \$0EEDFACE. 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 C++Builder exception object.

If a DLL does not use the SysUtils unit, C++Builder's exception support is disabled. In that case, if a run-time error occurs in a DLL, the application that called the DLL terminates. Because a DLL has no way of knowing whether it was called from an Object Pascal application or an application written in another programming language, it is not possible for the DLL to invoke the application's exit procedures before the application is terminated. The application is simply aborted and removed from memory. For this reason, make sure that there are sufficient checks in any DLL code so such errors do not occur.

See also

[Handling exceptions](#)

[Writing DLLs](#)

The Shared Memory Manager

Writing DLLS

If a DLL exports any procedures or functions that pass long strings 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 module (application or DLL) allocates memory with New or GetMem which is deallocated by a call to Dispose or FreeMem in another module.

ShareMem is the interface unit for the C++BuilderMM.DLL shared memory manager, which must be deployed along with applications and/or libraries that use ShareMem. When an application or DLL uses ShareMem, the application or DLL's memory manager is replaced by the memory manager in C++BuilderMM.DLL, making it possible to share dynamically allocated memory between multiple modules.

When used by an application or library, the ShareMem unit should be the first unit listed in an application or library's **uses** clause.

The DLLProc variable

See also [Writing DLLs](#)

The DLLProc variable defined in the System unit allows a DLL to monitor all calls that the operating system makes to the DLL's entry point. This functionality is normally only of interest to DLLs that support multi-threading.

To monitor operating system calls to a DLL's entry point, assign the address of a procedure with the following parameter list to the DLLProc variable.

```
procedure DLLHandler(Reason: Integer);
```

When the DLL procedure is invoked, the Reason parameter will contain one of the following values (defined in the Windows unit):

DLL_PROCESS_DETACH	Indicates that the DLL is detaching from the address space of the calling process as a result of either a clean process exit or of a call to FreeLibrary.
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread.
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly.

See also

Writing robust applications

C++Builder provides you with a mechanism to ensure that your applications are robust, meaning that they handle errors in a consistent manner that allows the application to recover from errors if possible and to shut down if need be, without losing data or resources.

Error conditions in C++Builder are indicated by exceptions.

To use exceptions to create safe applications, you need to understand the following tasks:

- [Protecting blocks of code](#)
- [Protecting resource allocations](#)
- [Handling RTL exceptions](#)
- [Handling component exceptions](#)
- [Silent exceptions](#)
- [Defining your own exceptions](#)

Protecting blocks of code

[See also](#)

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application will present a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

To protect blocks of code you need to understand

- [Responding to exceptions](#)
- [Exceptions and the flow of execution](#)
- [Nesting exception responses](#)

See also

[Protecting resource allocations](#)

[Handling RTL exceptions](#)

[Handling component exceptions](#)

[Silent exceptions](#)

[Defining your own exceptions](#)

Responding to exceptions

[See also](#)

When an error condition occurs, the application raises an exception, meaning it creates an exception object. Once an exception is raised, your application can execute cleanup code, handle the exception, or both.

Executing cleanup code

The simplest way to respond to an exception is to guarantee that some cleanup code is executed. This kind of response doesn't correct the condition that caused the error but lets you ensure that your application doesn't leave its environment in an unstable state.

You typically use this kind of response to ensure that the application frees allocated resources, regardless of whether errors occur.

Handling the exception

Handling an exception means making a specific response to a specific kind of exception. This clears the error condition and destroys the exception object, which allows the application to continue execution.

You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are easy to correct and retry, while others, such as running out of memory, might be more difficult for the application or the user to correct.

See also

[Raising an exception](#)

[Nesting exception responses](#)

[Protecting resource allocations](#)

[Handling RTL exceptions](#)

[Handling component exceptions](#)

[Silent exceptions](#)

[Defining your own exceptions](#)

Exceptions and the flow of execution

[See also](#)

[Example](#)

Object Pascal makes it easy to incorporate error handling into your applications because exceptions don't get in the way of the normal flow of your code. In fact, by moving error checking and error handling out of the main flow of your algorithms, exceptions can simplify the code you write.

When you declare a protected block, you define specific responses to exceptions that might occur within that block. When an exception occurs in that block, execution immediately jumps to the response you defined, then leaves the block.

Example

Here's some code that includes a protected block. If any exception occurs in the protected block, execution jumps to the exception-handling part, which beeps. Execution resumes outside the block.

```
...
try { begin the protected block }
    Font.Name := 'Courier'; { if any exception occurs... }
    Font.Size := 24; { ...in any of these statements... }
    Color := clBlue;
except { ...execution jumps to here }
    on Exception do MessageBeep(0);{ this handles any exception by beeping }
end;
... { execution resumes here, outside the protected block}
```

See also

[Responding to exceptions](#)

[Nesting exception responses](#)

Nesting exception responses

[See also](#)

Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks inside other blocks, you can customize responses even within blocks that already customize responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:

You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:

You can also mix different kinds of exception-response blocks, nesting resource protections within exception handling blocks and vice versa.

See also

[Responding to exceptions](#)

[Exceptions and the flow of execution](#)

[Scope of exception handlers](#)

Protecting resource allocations

[See also](#)

One key to having a robust application is ensuring that if it allocates resources, it also releases them, even if an exception occurs. For example, if your application allocates memory, you need to make sure it eventually releases the memory, too. If it opens a file, you need to make sure it closes the file later.

Keep in mind that exceptions don't come just from your code. A call to an RTL routine, for example, or another component in your application might raise an exception. Your code needs to ensure that if these conditions occur, you release allocated resources.

To protect resources effectively, you need to understand the following:

- [What kind of resources need protection?](#)
- [Creating a resource-protection block](#)

See also

[Protecting blocks of code](#)

[Handling RTL exceptions](#)

[Handling component exceptions](#)

[Silent exceptions](#)

[Defining your own exceptions](#)

What kind of resources need protection?

[See also](#)

[Example](#)

Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are

- Files
- Memory
- Windows resources
- Objects

Example

The following event handler allocates memory, then generates an error, so it never executes the code to free the memory:

```
procedure TForm1.Button1Click(Sender: TComponent);  
var  
    APointer: Pointer;  
    AnInteger, ADividend: Integer;  
begin  
    ADividend := 0;  
    GetMem(APointer, 1024); { allocate 1K of memory }  
    AnInteger := 10 div ADividend; { this generates an error }  
    FreeMem(APointer, 1024); { it never gets here }  
end;
```

Although most errors are not that obvious, the example illustrates an important point: When the division-by-zero error occurs, execution jumps out of the block, so the FreeMem statement never gets to free the memory.

In order to guarantee that the FreeMem gets to free the memory allocated by GetMem, you need to put the code in a resource-protection block.

See also

[Creating a resource-protection block](#)

Creating a resource-protection block

See also [Example](#)

To ensure that you free allocated resources, even in case of an exception, you embed the resource-using code in a protected block, with the resource-freeing code in a special part of the block. Here's an outline of a typical protected resource allocation:

```
{ allocate the resource }  
try  
  { statements that use the resource }  
finally  
  { free the resource }  
end;
```

The key to the **try..finally** block is that the application always executes any statements in the **finally** part of the block, even if an exception occurs in the protected block. When any code in the **try** part of the block (or any routine called by code in the **try** part) raises an exception, execution immediately jumps to the **finally** part, which is called the cleanup code. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the **try** part.

The statements in the termination code do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the termination code.

Note: A resource-protection block doesn't handle the exception. In fact, the termination code doesn't have information about whether an exception even occurred, so it can't determine whether it needs to handle an exception. If an exception occurs in a resource-protection block, execution first goes to the termination code, then leaves the block with the exception still raised. The block that contains the protected block can then respond to the exception.

Example

Here's an event handler that allocates memory and generates an error, but still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TComponent);  
var  
    APointer: Pointer;  
    AnInteger, ADividend: Integer;  
begin  
    ADividend := 0;  
    GetMem(APointer, 1024); { allocate 1K of memory }  
    try  
        AnInteger := 10 div ADividend;      { this generates an error }  
    finally  
        FreeMem(APointer, 1024);          { execution resumes here, despite the  
error }  
    end;  
end;
```

See also

[What kind of resources need protection?](#)

Handling RTL exceptions

[See also](#)

When you write code that calls routines in the run-time library (RTL), such as mathematical functions or file-handling procedures, the RTL reports errors back to your application in the form of exceptions. By default, RTL exceptions generate a message that the application displays to the user. You can define your own exception handlers to handle RTL exceptions in other ways.

There are also [silent exceptions](#) that do not, by default, display a message.

To handle RTL exceptions effectively, you need to understand the following:

- [What are the RTL exceptions?](#)
- [Creating an exception handler](#)
- [Providing default exception handlers](#)
- [Handling classes of exceptions](#)
- [Reraising the exception](#)

See also

[Protecting blocks of code](#)

[Protecting resource allocations](#)

[Handling component exceptions](#)

[Silent exceptions](#)

[Defining your own exceptions](#)

What are the RTL exceptions?

[See also](#)

The run-time library's exceptions are defined in the SysUtils unit, and they all descend from a generic exception-object type called Exception. Exception provides the string for the message that RTL exceptions display by default.

There are seven kinds of exceptions raised by the RTL:

- [Input/output exceptions](#)
- [Heap exceptions](#)
- [Integer math exceptions](#)
- [Floating-point math exceptions](#)
- [Typecast exceptions](#)
- [Conversion exceptions](#)
- [Hardware exceptions](#)

Input/output exceptions

Input/output (I/O) exceptions can sometimes occur when the RTL tries to access files or I/O devices. Most I/O exceptions are related to error codes returned by Windows or DOS when accessing a file.

The SysUtils unit defines a generic input/output exception called EInOutError that contains an object field named ErrorCode that indicates what error occurred. You can access that field in the exception-object instance to determine how to handle the exception.

Heap exceptions

Heap exceptions can sometimes occur when you try to allocate or access dynamic memory. The SysUtils unit defines two heap exceptions called EOutOfMemory and EInvalidPointer. The following table shows the specific heap exceptions, each of which descends directly from Exception:

Exception	Meaning
EOutOfMemory	There was not enough space on the heap to complete the requested operation.
EInvalidPointer	The application tried to dispose of a pointer that points outside the heap. Usually, this means the pointer was already disposed of.

Integer math exceptions

Integer math exceptions can occur when you perform operations on integer-type expressions. The SysUtils unit defines a generic integer math exception called EIntError. The RTL never raises an EIntError, but it provides a base from which all the specific integer math exceptions descend.

The following table shows the specific integer math exceptions, each of which descends directly from EIntError.

Exception	Meaning
EDivByZero	Attempt to divide by zero.
ERangeError	Number or expression out of range.
EIntOverflow	Integer operation overflowed.

Floating-point math exceptions

Floating-point math exceptions can occur when you perform operations on real-type expressions. The SysUtils unit defines a generic floating-point math exception called EMathError. The RTL never raises an EMathError, but it provides a base from which all the specific floating-point math exceptions descend.

The following table shows the specific floating-point math exceptions, each of which descends directly from EMathError:

Exception	Meaning
EInvalidOp	Processor encountered an undefined instruction.
EZeroDivide	Attempt to divide by zero.
EOverflow	Floating-point operation overflowed.
EUnderflow	Floating-point operation underflowed.

Typecast exceptions

Typecast exceptions can occur when you attempt to typecast an object into another type using the as operator. The SysUtils unit defines an exception called EInvalidCast that the RTL raises when the requested typecast is illegal.

Conversion exceptions

Conversion exceptions can occur when you convert data from one form to another using functions such as `IntToStr`, `StrToInt`, `StrToFloat`, and so on. The `SysUtils` unit defines an exception called `EConvertError` that the RTL raises when the function cannot convert the data passed to it.

Hardware exceptions

Hardware exceptions can occur in two kinds of situations: either the processor detects a fault it can't handle, or the application intentionally generates an interrupt to break execution. Hardware exception-handling is not compiled into DLLs, only into standalone applications.

The SysUtils unit defines a generic hardware exception called `EProcessorException`. The RTL never raises an `EProcessorException`, but it provides a base from which the specific hardware exceptions descend.

The following table shows the specific hardware exceptions.

Exception	Meaning
<code>EFault</code>	Base exception object from which all fault objects descend.
<code>EGPFault</code>	General protection fault, usually caused by an uninitialized pointer or object.
<code>EStackFault</code>	Illegal access to the processor's stack segment.
<code>EPageFault</code>	The Windows memory manager was unable to correctly use the swap file.
<code>EInvalidOpCode</code>	Processor encountered an undefined instruction. This usually means the processor was trying to execute data or uninitialized memory.
<code>EBreakpoint</code>	The application generated a breakpoint interrupt.
<code>ESingleStep</code>	The application generated a single-step interrupt.

You should rarely encounter the fault exceptions, other than the general protection fault, because they represent serious failures in the operating environment. The breakpoint and single-step exceptions are generally handled by the integrated debugger within `C++Builder`.

See also

[Creating an exception handler](#)

[Providing default exception handlers](#)

[Handling classes of exceptions](#)

[Reraising the exception](#)

Creating an exception handler

[See also](#) [Example](#)

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code.

To define an exception handler, embed the code you want to protect in an exception-handling block and specify the exception handling statements in the **except** part of the block. Here is an outline of a typical exception-handling block:

```
try
  { statements you want to protect }
except
  { exception-handling statements }
end;
```

The application executes the statements in the **except** part only if an exception occurs during execution of the statements in the **try** part. Execution of the **try** part statements includes routines called by code in the **try** part. That is, if code in the **try** part calls a routine that doesn't define its own exception handler, execution returns to the exception-handling block, which handles the exception.

When a statement in the **try** part raises an exception, execution immediately jumps to the **except** part, where it steps through the specified exception-handling statements, or exception handlers, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

See also

[Using the exception instance](#)

[Scope of exception handlers](#)

[Providing default exception handlers](#)

[Handling classes of exceptions](#)

[Reraising the exception](#)

Exception-handling statements

[See also](#) [Example](#)

Each statement in the **except** part of a **try..except** block defines code to execute to handle a particular kind of exception. The form of these exception-handling statements is as follows:

```
on <type of exception> do <statement>;
```

By using exceptions, you can spell out the "normal" expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every single time to make sure you're allowed to proceed with each step in the calculation.

Example

The following example defines an exception handler for division by zero to provide a default result:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;  
begin  
  try  
    Result := Sum div NumberOfItems;  
  except  
    on EDivByZero do Result := 0;  
  end;  
end;
```

Note that this is clearer than having to test for zero every time you call the function. Here's an equivalent function that doesn't take advantage of exceptions:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;  
begin  
  if NumberOfItems <> 0 then  
    Result := Sum div NumberOfItems  
  else Result := 0;  
end;
```

The difference between these two functions really defines the difference between programming with exceptions and programming without them. This example is quite simple, but you can imagine a more complex calculation involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid.

See also

[Using the exception instance](#)

[Scope of exception handlers](#)

Using the exception instance

[See also](#) [Example](#)

Most of the time, an exception handler doesn't need any information about an exception other than its type, so the statements following **on..do** are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of **on..do** that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance.

The temporary variable (E in this example) is of the type specified after the colon (EInvalidOperation in this example). You can use the as operator to typecast the exception into a more specific type if needed.

Note: Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating a fatal application error.

Example

If you create a new project that contains a single form, you can add a scroll bar and a command button to the form. Double-click the button and add the following line to its click-event handler:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

That line raises an exception because the maximum value of a scroll bar must always exceed the minimum value. The default exception handler for the application opens a dialog box containing the message in the exception object. You can override the exception handling in this handler and create your own message box containing the exception's message string:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK],
0);
end;
```

See also

[Exception-handling statements](#)

[Scope of exception handlers](#)

Scope of exception handlers

[See also](#)

You don't have to provide handlers for every kind of exception in every block. In fact, you need to provide handlers only for those exceptions you want to handle specially within that particular block.

If a block doesn't handle a particular exception, execution leaves that block and returns to the block that contains the block (or to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until.

See also

[Exception-handling statements](#)

[Using the exception instance](#)

Providing default exception handlers

[See also](#)

You can provide a single default exception handler to handle any exceptions you haven't provided specific handlers for. To do that, you add an **else** part to the **except** part of the exception-handling block:

```
try
  { statements }
except
  on ESomething do { specific exception-handling code };
  else { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from the containing block.

Warning!: You should probably never use this all-encompassing default exception handler. The **else** clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. In other cases, it's better to execute cleanup code and leave the handling to code that has more information about the exception and how to handle it.

See also

[Creating an exception handler](#)

[Handling classes of exceptions](#)

[Reraising the exception](#)

Handling classes of exceptions

[See also](#) [Example](#)

Because exception objects are part of a hierarchy, you can specify handlers for entire parts of the hierarchy by providing a handler for the exception class from which that part of the hierarchy descends.

You can still specify specific handlers for more specific exceptions, but you need to place those handlers above the generic handler, because the application searches the handlers in the order they appear in, and executes the first applicable handler it finds.

Example

The following block outlines an example that handles all integer math exceptions specially:

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

Note that if the handler for EIntError came before the handler for ERangeError, execution would never reach the specific handler for ERangeError.

See also

[Creating an exception handler](#)

[Providing default exception handlers](#)

[Reraising the exception](#)

Reraising the exception

[See also](#) [Example](#)

Sometimes when you handle an exception locally, you actually want to augment the handling in the enclosing block, rather than replacing it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond.

When an exception occurs, you might want to display some sort of message to the user, then proceed with the standard handling. To do that, you declare a local exception handler that displays the message then calls the reserved word **raise**.

If code in the { statements } part raises an exception, only the handler in the outer **except** part executes. However, if code in the { special statements } part raises an exception, the handling in the inner **except** part is executed, followed by the more general handling in the outer **except** part.

By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

Example

The following example reraises an exception:

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
      begin
        { handling for only the special statements }
        raise;    { reraise the exception }
      end;
    end;
except
  on ESomething do ...;    { handling you want in all cases }
end;
```

See also

[Creating an exception handler](#)

[Providing default exception handlers](#)

[Handling classes of exceptions](#)

Handling component exceptions

[See also](#) [Example](#)

C++Builder's components raise exceptions to indicate error conditions. Most component exceptions indicate programming errors that would otherwise generate a run-time error. The mechanics of handling component exceptions are no different than [handling RTL exceptions](#).

A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises an "Index out of range" exception.

Example

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ListBox1.Items.Add('a string');{ add a string to list box }  
    ListBox1.Items.Add('another string');{ add another string... }  
    ListBox1.Items.Add('still another string');{ ...and a third string }  
    try  
        Caption := ListBox1.Items[3];{ set form caption to fourth string in  
list box }  
    except  
        on EListError do  
            MessageDlg('List box contains fewer than four strings', mtWarning,  
[mbOK], 0);  
    end;  
end;
```

If you click the button once, the list box has only three strings, so accessing the fourth string (Items[3]) raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

See also

[Protecting blocks of code](#)

[Protecting resource allocations](#)

[Handling RTL exceptions](#)

[Defining your own exceptions](#)

Silent exceptions

[See also](#) [Example](#)

C++Builder applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to handle an exception, but you want to abort an operation. Aborting an operation is similar to using the Break or Exit procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type EAbort. The default exception handler for C++Builder applications displays the error-message dialog box for all exceptions that reach it except those descended from EAbort.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the Abort procedure. Abort automatically raises an EAbort exception, which will break out of the current operation without displaying an error message.

Example

The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's OnClick event:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    I: Integer;  
begin  
    for I := 1 to 10 do{ loop ten times }  
    begin  
        ListBox1.Items.Add(IntToStr(I));    { add a numeral to the list }  
        if I = 7 then Abort;    { abort after the seventh one }  
    end;  
end;
```

See also

[Protecting blocks of code](#)

[Protecting resource allocations](#)

[Handling RTL exceptions](#)

[Handling component exceptions](#)

[Defining your own exceptions](#)

Defining your own exceptions

[See also](#)

In addition to protecting your code from exceptions generated by the run-time library and various components, you can use the same mechanism to manage exceptional conditions in your own code.

To use exceptions in your code, you need to understand these steps:

- [Declaring an exception object type](#)
- [Raising an exception](#)

See also

[Protecting blocks of code](#)

[Protecting resource allocations](#)

[Handling RTL exceptions](#)

[Handling component exceptions](#)

[Silent exceptions](#)

Declaring an exception object type

[See also](#) [Example](#)

Because exceptions are objects, defining a new kind of exception is as simple as declaring a new object type. Although you can raise any object instance as an exception, the standard exception handlers handle only exceptions descended from `Exception`.

It's therefore a good idea to derive any new exception types from `Exception` or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by a specific exception handler for that exception, one of the standard handlers will handle it instead.

Example

For example, consider the following declaration:

type

```
EMyException = class(Exception);
```

If you raise EMyException but don't provide a specific handler for EMyException, a handler for Exception (or a default exception handler) will still handle it. Because the standard handling for Exception displays the name of the exception raised, you could at least see that it was your new exception raised.

See also

[Raising an exception](#)

Raising an exception

[See also](#)

[Example](#)

To indicate an error condition in an application, you can raise an exception which involves constructing an instance of that type and calling the reserved word **raise**.

To raise an exception, call the reserved word **raise**, followed by an instance of an exception object.

When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

Setting the exception address

Raising an exception sets the ErrorAddr variable in the System unit to the address where the application raised the exception. You can refer to ErrorAddr in your exception handlers, for example, to notify the user of where the error occurred. You can also specify a value for ErrorAddr when you raise an exception.

To specify an error address for an exception, add the reserved word **at** after the exception instance, followed by an address expression such as an identifier.

Examples

For example, given the following declaration,

```
type  
  EPasswordInvalid = class(Exception);
```

you can raise a "password invalid" exception at any time by calling **raise** with an instance of EPasswordInvalid, like this:

```
if Password <> CorrectPassword then  
  raise EPasswordInvalid.Create('Incorrect password entered');
```

See also

[Declaring an exception object type](#)

[Reraising the exception](#)

Scope

[See also](#)

[Example](#)

[Language definition](#)

Scope of an [identifier](#) within a program or unit. defines whether or not that identifier can be used by other procedures and functions in the program or unit.

Scope can either be local or global. Local identifiers are only visible to those routines and declarations contained within the block which declares the identifier.

Global identifiers are declared within the interface section of a unit and are visible to all routines and declaration within that unit.

When designing the structure of your program, follow these three rules of scope:

- Each identifier has meaning only within the block in which it is declared, and only after the point in that block at which it is declared.
- If a global identifier is redefined within a block, then the innermost (most deeply nested) definition takes precedence from the point of declaration until the end of the block.
- When invoking procedures recursively, a reference to a global variable always refers to the instance of the variable in the most recent invocation of the procedure in which that variable is defined.

There are several different types of scope, they are:

[Block scope](#)

[Component scope](#)

[Component visibility](#)

[Unit scope](#)

[Scope of interface and standard identifiers](#)

See also

[Global and local variables](#)

[Rules of scope](#)

Component scope

[See also](#) [Scope](#)

The scope of a component identifier declared in a class type extends from the point of declaration to the end of the class type definition, and extends over all descendants of the class type and the blocks of all method declarations of the class type. Also, the scope of component identifiers includes field, method, and property designators, and **with** statements that operate on variables of the given class type.

A component identifier declared in a class type can be redeclared in the block of a method declaration of the class type. In that case, the `Self` parameter can be used to access the component whose identifier was redeclared.

A component identifier declared in an ancestor class type can be redeclared in a descendant of the class type. Such redeclaration effectively hides the inherited component, although the **inherited** keyword can be used to bring the inherited component back into scope.

See also

[Block scope](#)

[Rules of scope](#)

[Scope of interface and standard identifiers](#)

[Unit scope](#)

Block scope

[See also](#)

[Example](#)

[Scope](#)

In a block, the scope of an identifier or label is from the point of declaration to the end of the current block, and includes all nested blocks.

If you override an identifier within a nested block, the scope of the new identifier is only within the nested block and it does not extend outward.

The scope of a type identifier is local to the block in which the type declaration occurs. Except for pointer types, the declaration does not include itself.

Example

```
program Outer;    { Start of outer scope }  
type  
  I = Integer;    { Define I as type Integer }  
var  
  T: I;          { Define T as an Integer variable }  
procedure Inner; { Start of inner scope }  
type  
  T = I;        { redefine T as type Integer }  
var  
  I: T;         { Redefine I as an Integer variable }  
begin  
  I := 1;  
end;          { End of inner scope }  
begin  
  T := !;  
end.         { End of outer scope }
```


See also

Unit scope

Rules of scope

Record scope

See also [Scope](#)

The scope of a field identifier declared in a record-type definition extends from the point of declaration to the end of the record type definition.

The scope of field identifiers includes field designators and with statements that operate on variable references of the given record.

See also

[Record types](#)

[Rules of scope](#)

Unit scope

[See also](#) [Scope](#)

An identifier declared in the interface part of a unit is available to other programs or units that specify, in their uses clauses, that unit containing the identifier.

When more than one unit is listed in the **uses** clause, the following rules apply:

- The scope of each unit includes all the units that follow it, and the program (or unit) that contains the **uses** clause.
 - The first unit defines the outermost scope, and the last unit defines the innermost scope.
- Therefore, if an identifier is declared in more than one unit, an unqualified reference to the identifier selects the instance declared by the last unit. To specify an instance declared in any other unit, use a qualified identifier.

Note: The scope of the System unit is global so every program has access to the Object Pascal standard identifiers. The System unit doesn't need to be listed in the uses clause.

See also

Qualifiers

Rules of scope

Example

```
program scope2;
var
  A: integer;      {Global variable}

procedure SetA;
var
  A : integer;    {Creates local variable A}
begin
  A := 4
end;              {Destroys local variable A}

begin
  A := 3;         {Assigns value to global variable A}
  SetA;          {Calls procedure SetA}
  Writeln(A)     {Value of A = 3 -- not 4!}
end.
```

Rules of scope

See also [Scope](#)

The presence of an identifier or label in a declaration defines the identifier or label, and each time the identifier or label occurs again, it must be within the scope of this declaration.

The scope of an identifier or label extends from its declaration to the end of the current block, including all blocks enclosed by the current block.

The scope also extends over all descendants of the object type, including field designators and with statements that operate on variable references to the given object type.

The following three rules are the exceptions

1. Redeclaration in an enclosed block:

Suppose that Exterior is a block that encloses another block, Interior. If Exterior and Interior both have an identifier with the same name (for example, J), Interior can only access the J it declared, and Exterior can only access the J it declared.

2. Position of declaration within its block:

Identifiers and labels cannot be used until after they are declared.

An identifier or label's declaration must come before any occurrence of that identifier or label in the program text, unless it is the base type of a pointer type that has not yet been declared. However, the identifier must eventually be declared in the same type declaration part that the pointer type occurs in.

3. Redeclaration within a block:

An identifier or label can only be declared once in the outer level of a given block, unless it is declared within a contained block or is in a record's field list.

A record field identifier is declared within a record type, and is significant only in combination with a reference to a variable of that record type.

You can redeclare a field identifier (with the same spelling) within the same block, but not at the same level within the same record type.

However, an identifier that has been declared can be redeclared as a field identifier in the same block.

See also

[Block scope](#)

[Interface and standard identifier scope](#)

[Component scope](#)

[Record scope](#)

[Unit scope](#)

Interface and standard identifier scope

See also [Scope](#)

Programs or units containing **uses** clauses have access to identifiers belonging to the interface parts of the units listed in those **uses** clauses.

Each unit 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, the last unit represents the innermost scope.

If two or more units declare the same identifier, an unqualified reference to the identifier will select the instance declared by the last unit in the **uses** clause.

See also

[Block scope](#)

[Component scope](#)

[Rules of scope](#)

[Unit scope](#)

[Uses clause](#)

Compiler directives: definition and use

See also [Compiler directives](#)

Compiler directives enable you to customize the default behavior of the compiler. Compiler directives are comments with a special syntax, and can be used wherever comments are allowed. Their scope can be local or global, but not all directives can be used in both contexts.

- Local directives can appear anywhere in a program unit; they affect only part of the compilation.
- Global directives must appear before the declaration part of the program or unit being compiled; they affect the entire compilation.

Compiler directives fall into the following three categories:

Switch directives

Switch directives enable or disable compiler features.

For the single-letter versions, you add either + or - immediately after the directive letter.

For the long version, you supply the word "on" or "off."

You can group multiple switch directives, separating them with commas (and no spaces). For example:

```
{ $F+, R+, D- }
```

Parameter directives

Parameter directives pass information to the compiler such as a file name, text, or memory sizes. There must be at least one blank space between the directive name and its parameters. For example:

```
{ $I TYPES.INC }  
{ $L YOUR.DOC }
```

Conditional directives and symbols

Conditional directives control compilation of parts of the source text, based on evaluation of a symbol following the directive. You can define your own symbols or you can use the Object Pascal predefined symbols. Conditional directives must be specified within code.

See also

[Conditional directives and symbols](#)

[Using conditional directives](#)

The predefined conditional symbols are: `CONSOLE`, `WIN32`, `CPU386`, and `VER90`.

Alphabetic list of compiler directives

[See also](#)

This topic provides an alphabetic list of all compiler directives supported by C++Builder. From the list, you can jump to a complete definition of each directive.

Directive	Type	Description
<u>\$A</u>	Switch	Align Data
<u>\$ALIGN</u>	Switch	Align Data
<u>\$APPTYPE</u>	Parameter	Application type
<u>\$B</u>	Switch	Boolean Evaluation
<u>\$BOOLEVAL</u>	Switch	Boolean Evaluation
<u>\$D</u>	Switch	Debug Information
<u>\$DEBUGINFO</u>	Switch	Debug Information
<u>\$D Text</u>	Parameter	Description
<u>\$DESCRIPTION Text</u>	Parameter	Description
<u>\$EXTENDEDSYNTAX</u>	Switch	Extended Syntax
<u>\$H</u>	Switch	Long Strings
<u>\$HINTS</u>	Switch	Compiler Hints
<u>\$I</u>	Switch	Input/Output Checking
<u>\$I FileName</u>	Parameter	Include File
<u>\$IMAGEBASE</u>	Parameter	Code-Image Base Address
<u>\$INCLUDE FileName</u>	Parameter	Include File
<u>\$IOCHECKS</u>	Switch	Input/Output Checking
<u>\$J</u>	Switch	Writeable typed constants
<u>\$L</u>	Switch	Local Symbol Information
<u>\$L FileName</u>	Parameter	Link Object File
<u>\$LINK FileName</u>	Parameter	Link Object File
<u>\$LOCALSYMBOLS</u>	Switch	Local Symbol Information
<u>\$LONGSTRINGS</u>	Switch	Long Strings
<u>\$M</u>	Switch	Run-time Type Information
<u>\$M StackSize</u>	Parameter	Maximum Stack Size
<u>\$MAXSTACKSIZE StackSize</u>	Parameter	Maximum Stack Size
<u>\$MINENUMSIZE</u>	Switch/Parameter	Enumerated Type Size
<u>\$MINSTACKSIZE StackSize</u>	Parameter	Minimum Stack Size
<u>\$O</u>	Switch	Optimization
<u>\$OPENSTRINGS</u>	Switch	Open String Parameters
<u>\$OPTIMIZATION</u>	Switch	Optimization
<u>\$OVERFLOWCHECKS</u>	Switch	Arithmetic Overflow Checking
<u>\$P</u>	Switch	Open String Parameters
<u>\$Q</u>	Switch	Arithmetic Overflow Checking
<u>\$R</u>	Switch	Range Checking
<u>\$R FileName</u>	Parameter	Resource File

<u>\$RANGECHECKS</u>	Switch	Range Checking
<u>\$REFERENCEINFO</u>	Switch	Symbol Reference Information
<u>\$RESOURCE</u>	FileName Parameter	Resource File
<u>\$SAFEDIVIDE</u>	Switch	Pentium-safe FDIV operations
<u>\$STACKFRAMES</u>	Switch	Windows Stack Frame
<u>\$T</u>	Switch	Typed @ Operator
<u>\$TYPEDADDRESS</u>	Switch	Typed @ Operator
<u>\$TYPEINFO</u>	Switch	Run-time Type Information
<u>\$U</u>	Switch	Pentium-safe FDIV operations
<u>\$V</u>	Switch	Var-String Checking
<u>\$VARSTRINGCHECKS</u>	Switch	Var-String Checking
<u>\$W</u>	Switch	Windows Stack Frame
<u>\$WARNINGS</u>	Switch	Compiler Warnings
<u>\$WRITEABLECONST</u>	Switch	Writeable typed constants
<u>\$X</u>	Switch	Extended Syntax
<u>\$Y</u>	Switch	Symbol Reference Information
<u>\$Z</u>	Switch	Word Size Enumerated Types

See also

[Conditional directives and symbols](#)

[Definition and use of compiler directives](#)

[Using conditional directives](#)

Conditional directives and symbols

[See also](#)

Conditional directives control compilation of parts of the source text, based on evaluation of a symbol following the directive. You can define your own symbols or you can use the Object Pascal predefined symbols.

Conditional directive	Meaning
------------------------------	----------------

<u>\$DEFINE</u>	Defines a conditional symbol
<u>\$ELSE</u>	Compiles or ignores a portion of source text
<u>\$ENDIF</u>	Ends the conditional section
<u>\$IFDEF</u>	Compiles source text if Name is defined
<u>\$IFNDEF</u>	Compiles source text if Name is NOT defined
<u>\$IFOPT</u>	Compiles source text if a compiler switch is in a specified state (+ or -)
<u>\$UNDEF</u>	Undefines a previously defined conditional symbol

Conditional symbol	Meaning
---------------------------	----------------

<u>CONSOLE</u>	Application is being compiled as a console application.
<u>CPU386</u>	CPU is an Intel 386 or better.
<u>WIN32</u>	The operating environment is the Win32 API.
<u>VER90</u>	Compiles based on the version specified. For example, code preceded by { <u>\$IFDEF</u> VER90} compiles only if the compiler version is 9.0.

See also

[Alphabetic list of compiler directives](#)

[Definition and use of compiler directives](#)

[Using conditional directives](#)

Using conditional directives

See also [Compiler directives](#)

Conditional directives produce different code from the same source text, based on the state of conditional symbols. Object Pascal identifiers cannot be used in conditional directives.

Note: Changing your conditional defines should generally be followed by rebuilding your program.

There are two possible conditional constructs:

- `{ $IFxxx } ... { $ENDIF }`
- `{ $IFxxx } ... { $ELSE } ... { $ENDIF }`

\$IF ... \$ENDIF

The `$IFxxx ... $ENDIF` construct compiles the source code between `$IFxxx` and `$ENDIF` only if the condition specified in `$IFxxx` evaluates to True.

If the condition is False, the source text between the two directives is ignored.

\$IF ... \$ELSE ... \$ENDIF

The source code in a `$IFxxx ... $ELSE ... $ENDIF` construct compiles when the following conditions apply:

- If `$IFxxx` is True, the source text between `$IFxxx` and `$ELSE` compiles.
- If `$IFxxx` is False, the source text between `$ELSE` and `$ENDIF` compiles.

Conditional constructs can be nested to 16 levels deep.

Each `$IFxxx` must have a matching `$ENDIF`.

\$IFDEF

Compiles the source text that follows it if Name is defined.

Syntax

```
{ $IFDEF Name }
```

Within a portion of source text delimited by an `$IFDEF` (or an `$IFNDEF`) and an `$ENDIF`, `$ELSE` compiles the source code that follows it if the `$IFDEF` (or `$IFNDEF`) condition is false.

If the `$IFDEF` (or `$IFNDEF`) condition is met, `$ELSE` ignores the source code that follows it.

\$IFNDEF

Compiles the source text that follows it if Name is not defined.

Syntax

```
{ $IFNDEF Name }
```

\$IFOPT

Compiles the source text that follows it if `switch` is currently in the specified state.

Syntax

```
{ $IFOPT Switch }
```

Switch is the name of a switch option, followed by `+` or `-`:

```
Switch+   Switch is On
```

```
Switch-   Switch is Off
```

See also

[\\$DEFINE and \\$UNDEF Conditional Symbol Directives](#)

\$DEFINE and \$UNDEF conditional symbol directives

See also [Compiler directives](#)

Use \$DEFINE and \$UNDEF to define conditional symbols that the compiler evaluates before compiling. Conditional symbols are similar to Boolean variables in that they are either True or False. They follow Object Pascal identifier naming conventions but cannot be used in the actual program, just as Object Pascal identifiers cannot be used in conditional directives.

\$DEFINE

Defines a conditional symbol with the given Name (sets the symbol to True). The scope is within the current source file only; not globally across all source files. To define something across all modules, use the /D command-line option, or choose ** from the Options|Project|Conditional Defines dialog.

Syntax

```
{ $DEFINE Name }
```

Within the program, the compiler recognizes the defined symbol until the symbol appears in an \$UNDEF Name directive. \$DEFINE Name has no effect if Name is already defined.

\$UNDEF

Undefines a previously defined conditional symbol of Name (sets it to False).

Syntax

```
{ $UNDEF Name }
```

The symbol does not exist for the remainder of the compilation or until it reappears in a \$DEFINE directive. \$UNDEF Name directive has no effect if Name is already undefined.

See also

[Identifiers](#)

[Using conditional directives](#)

Predefined conditional symbols

See also [Compiler directives](#)

C++Builder defines the following conditional symbols.

CONSOLE

Defined if an application is being compiled as a console application.

CPU386

Indicates that the CPU is an Intel 386 or better.

VER90

Always defined, indicating that this is version 9.0 of the Object Pascal compiler. Each version has corresponding predefined symbols; for example, version 9.1 would have VER91 defined, version 9.5 would have VER95 defined, and so on.

WIN32

Indicates that the operating environment is the Win32 API.

See also

[Using conditional directives](#)

Align fields directive { \$A }, { \$ALIGN }

[Compiler directives](#)

This compiler directive controls alignment of fields in record types.

Syntax: { \$A+ } or { \$A- }
 { \$ALIGN ON } or { \$ALIGN OFF }

Default: { A+ }
 { \$ALIGN ON }

Scope: Local

Remarks

Regardless of the state of the \$A directive, variables and typed constants are always aligned for optimal access.

On { \$A+ }, { \$ALIGN ON }

When Align Fields is on, fields in record types that are declared without the packed modifier are aligned on a machine-word boundary (an even-numbered address). If required, unused bytes are inserted between variables to achieve word alignment.

This option does not affect byte-sized variables, fields of record structures or objects, or elements of arrays. A field in a record or object will align on a word boundary only if the total size of all fields before it is even. For every element of an array to align on a word boundary, the size of the elements must be even.

Off { \$A- }, { \$ALIGN OFF }

When Align Fields is off, no alignment measures are taken

Application type

[Compiler directives](#)

The **\$APPTYPE** directive controls whether to generate a Console or Graphical UI application.

Syntax: {\$APPTYPE GUI}
 {\$APPTYPE CONSOLE}

Default: {\$APPTYPE GUI}

Scope: Global

Graphical UI { \$APPTYPE GUI }

In the **{ \$APPTYPE GUI }** state, the compiler generates a graphical UI application. This is the normal state for a C++Builder application.

Console { \$APPTYPE CONSOLE }

In the **{ \$APPTYPE CONSOLE }** state, the compiler generates a console application. When a console application is started, Windows creates a text mode console window through which the user can interact with the application.

The Input and Output standard text files are automatically associated with the console window in a console application.

Remarks

The IsConsole Boolean variable declared in the System unit can be used to detect whether a program is running as a console or graphical UI application.

Note The **\$APPTYPE** directive is meaningful only in a program. It should not be used in a library or a unit.

Boolean evaluation directive { \$B }, { \$BOOLEVAL }

See also [Compiler directives](#)

Switches between the two different models of code generation for the **AND** and **OR** Boolean operators.

Syntax: {\$B+} or {\$B-}
 {\$BOOLEVAL ON} or {\$BOOLEVAL OFF}

Default: {\$B-}
 {\$BOOLEVAL OFF}

Scope: Local

On { \$B+ }, { \$BOOLEVAL ON }

When this option is on, the compiler generates code that evaluates every operand of a boolean expression built from the **AND** and **OR** operators, even when the result of the entire expression is already known.

Off { \$B- }, { \$BOOLEVAL OFF }

When this option is off, the compiler generates code for short-circuit boolean-expression evaluation.

This means evaluation stops as soon as the result of the entire expression becomes evident.

See also

[Boolean operators](#)

Debug information directive { \$D }, { \$DEBUGINFO }

See also [Compiler directives](#)

Enables or disables the generation of debug information.

Syntax: {\$D+} or {\$D-}
 {\$DEBUGINFO ON} or {\$DEBUGINFO OFF}

Default: {\$D+}
 {\$DEBUGINFO ON}

Scope: Global

Remarks

Debug information consists of a line-number table for each procedure which maps object code addresses into source text numbers.

Debug information increases the size of the unit files and takes up additional room when you compile programs that use the unit, but it does not affect the size or speed of the executable program. Debug information is recorded in the .DCU (unit) file, along with the unit's object code.

On { \$D+}, { \$DEBUGINFO ON }

When Debug Information is on, the compiler puts debug information into the unit (.DCU) file.

You can use the stand-alone or integrated debuggers to single-step and set breakpoints in modules compiled with Debug Information. When a run-time error occurs, the compiler can automatically go to the statement that caused the error.

The Project|Options|Linker|Map File radio buttons produce complete line information for a given module only if you've compiled that module with Debug Information.

The Debug Information directive is usually used with the [Local Symbols](#) directive.

If you want to use Turbo Debugger for Windows to debug your program, select Include TDW Debug Info from the Linker page of the Project Options dialog box, then recompile your program.

See also

[Debug and symbol information](#)

Using debug with symbol information switch directives

See also [Compiler directives](#)

The \$D, \$L and \$Y compiler directives are used together. \$L and \$Y can be thought of as subsets of \$D, with \$D having outermost scope, \$L the next in, and \$Y having innermost scope. The following table describes how these directives modify each other when used in combination.

Symbol	Result
{ \$D+, L-, Y+ }	Debugging information on all code in the interface section; no information on symbols in the implementation. (\$Y is ignored.)
{ \$D+, L-, Y- }	Debugging information on all code in the interface section; no information on symbols in the implementation. (\$Y is ignored.)
{ \$D-, L+, Y+ }	No debug information at all. (\$L, \$Y ignored.)
{ \$D+, L+, Y- }	No symbol reference or other ObjectBrowser information. This setting could save you some linking time.
{ \$D+, L+, Y+ }	Debug information on all symbols in the module; Line number and symbol information on local variables and types; Symbol cross-reference and other Browser information.

See also

[\\$D debug information](#)

[\\$L local symbol information](#)

[\\$Y symbol reference information](#)

Input/Output-Checking directive { \$I }, { \$IOCHECKS }

See also [Compiler directives](#)

The \$I directive enables or disables the automatic code generation that checks the result of a call to a file I/O procedure, such as Read, Write or Erase.

Syntax: {\$I+} or {\$I-}
 {\$IOCHECKS ON} or {\$IOCHECKS OFF}

Default: {\$I+}
 {\$IOCHECKS ON}

Scope: Local

On { \$I+ }, { \$IOCHECKS ON }

When I/O Checking is on, the compiler generates code to check for I/O errors after every I/O call. This is the result if a check fails:

All run-time errors halt your application.

Off { \$I- }, { \$IOCHECKS OFF }

When I/O Checking is off, you must use the IOResult function to check for I/O errors.

See also

[\\$! include file](#)

[Exception handling](#)

Writeable typed constants directive { \$J }, { \$WRITEABLECONST }

Compiler directives

The **\$J** directive controls whether typed constants can be modified.

Syntax: {\$J+} or {\$J-}
 {\$WRITEABLECONST ON} or {\$WRITEABLECONST OFF}

Default: {\$J-}
 {\$WRITEABLECONST OFF}

Scope: Local

On { \$J+ }, { \$WRITEABLECONST ON }

In the **{ \$J+ }** state, typed constants can be modified, and are in essence initialized variables.

Off { \$J- }, { \$WRITEABLECONST OFF }

In the **{ \$J- }** state, typed constants are truly constant, and any attempt to modify a typed constant causes the compiler to report an error.

Local symbol information directive { \$L }, { \$LOCALSYMBOLS }

See also [Compiler directives](#)

The \$L directive enables or disables the generation of local symbol information.

Syntax: {\$L+} or {\$L-}
 {\$LOCALSYMBOLS ON} or {\$LOCALSYMBOLS OFF}

Default: {\$L+}
 {\$LOCALSYMBOLS ON}

Scope: Global

Remarks

Local symbol information consists of the identifiers within the module's procedures and functions. Local symbol information does not include global variables or names declared in the interface section of a unit.

The Local Symbol Information directive is ignored if the [Debug Information directive](#) is off.

On { \$L+ }, { \$LOCALSYMBOLS ON }

When local symbols are on for a given program or unit, you can use the stand-alone or integrated debugger to examine and modify the module's local variables.

When the Map File option on the Linker Options page of the the Options|Project dialog box is selected, it produces local symbol information for a given module only if that module was compiled in the \$L+ state.

The local symbol information is recorded in the unit file, along with the unit's object code. Local symbol information increases the size of the unit files. It does not affect the size or speed of the executable program.

Off { \$L- }, { \$LOCALSYMBOLS OFF }

Disabling this option reduces the memory required to compile your program, makes the unit file smaller, and reduces the volume of debugger symbol information.

See also

[Debug and symbol information](#)

Open parameters directive { \$P }, { \$OPENSTRINGS }

[See also](#) [Example](#) [Compiler directives](#)

The \$P directive controls the meaning of variable parameters declared using the **string** keyword. Open parameters allow string variables of varying sizes to be passed to the same procedure or function.

Syntax: {\$P+} or {\$P-}
 {\$OPENSTRINGS ON} or {\$OPENSTRINGS OFF}

Default: {\$P-}
 {\$OPENSTRINGS OFF}

Scope: Global

On { \$P+ }, { \$OPENSTRINGS ON }

When Open Parameters is enabled, variable parameters declared using the **string** keyword are open string parameters. Regardless of the setting of this option, the OpenString identifier can always be used to declare open string parameters.

The actual parameter of an open-string parameter can be a variable of any string type, and within the procedure or function, the size attribute (maximum length) of the formal parameter will be the same as that of the actual parameter.

Open string parameters behave exactly as variable parameters of a string type, except that they cannot be passed as regular variable parameters to other procedures and functions.

Off { \$P- }, { \$OPENSTRINGS OFF }

When Open Parameters is off, Open parameters are disabled. In this state, variable parameters declared using the **string** keyword are normal variable parameters. This allows compatibility with earlier versions of Turbo Pascal.

Example

```
procedure MyProc (var S:string);  
begin  
    S:= 'abcdefghijk';  
end;
```

```
var  
shortstring: string[5];  
begin  
    MyProc (ShortString);  
end.
```

Compiler switch	Result
\$P-, V+	MyProc(ShortString) produces a compiler error of Type Mismatch.
\$P+, V-	MyProc(ShortString) is allowed, and the code generated ensures that assignments to S do not exceed the declared size of the actual parameter. After the call to MyProc, ShortString equals 'abcde'.
\$P-, V-	MyProc does not produce a compiler error, but might cause a memory overwrite error in your program, which could crash your system.

See also

[\\$V var-string checking](#)

Overflow checking directive { \$Q }, { \$OVERFLOWCHECKS }

See also [Compiler directives](#)

This compiler directive controls the generation of arithmetic overflow checking code.

Syntax: {\$Q+} or {\$Q-}
 {\$OVERFLOWCHECKS ON} or {\$OVERFLOWCHECKS OFF}

Default: {\$Q-}

Scope: Local

Remarks

An arithmetic overflow happens when the result of a calculation exceeds the size of the type of calculation. In some cases, information is lost.

On { \$Q+ }, { \$OVERFLOWCHECKS ON }

When Overflow Checking is on, the compiler generates code to check arithmetic overflow for the following integer operations:

\ + - * Abs Sqr Succ Pred Inc Dec

The code for each of these arithmetic operations is followed by additional code that verifies that the result is within the supported range.

If an overflow check fails, the program halts with a run-time error.

Enabling overflow checking slows down your program and makes it larger. Use it during program development and debugging and then turn it off when building your final product.

The \$Q directive is usually used in conjunction with the [\\$R directive](#).

Off { \$Q- }, { \$OVERFLOWCHECKS OFF }

When off, no arithmetic overflow checking is done.

See also

[Exception handling](#)

Range-checking directive { \$R }, { \$RANGECHECKS }

See also [Compiler directives](#)

This compiler directive enables and disables the generation of range-checking code

Syntax: {\$R+} or {\$R-}
 {\$RANGECHECKS ON} or {\$RANGECHECKS OFF}

Default: {\$R-}
 {\$RANGECHECKS OFF}

Scope: Local

On { \$R+ }, { \$RANGECHECKS ON }

When Range Checking is on, the compiler generates code to check that array and string subscripts are within bounds, and that assignments to scalar-type variables do not exceed their defined ranges. Range checking does not apply to Inc and Dec.

If a check fails, the program halts with a run-time error.

Enabling range-checking slows down your program and makes it larger. Enable this option during program development and debugging, and then turn it off when building your final product.

Off { \$R- }, { \$RANGECHECKS OFF }

When Range Checking is off, no range checking code is generated.

See also

[\\$Q arithmetic overflow checking](#)

[Exception handling](#)

Stack-overflow checking directive { \$S }, { \$STACKCHECKS }

[Compiler directives](#)

This compiler directive enables and disables the generation of stack-overflow checking code.

Syntax: {\$S+} or {\$S-}
 {\$STACKCHECKS ON} or {\$STACKCHECKS OFF}

Default: {\$S+}
 {\$STACKCHECKS ON}

Scope: Local

On { \$S+ }, { \$STACKCHECKS ON }

When Stack Overflow checking is on, the compiler generates code at the beginning of each procedure or function to check whether there is sufficient stack space for the local variables and other temporary storage.

Note: In general, Stack checking should be left on even in the final build of your program, unless you are certain your program will never overflow the stack.

Off { \$S- }, { \$STACKCHECKS OFF }

When Stack Checking is off, and there is not enough stack space available, a call to a procedure or function is likely to cause a system crash or halt with a run-time error.

Typed @ operator directive { \$T }, { \$TYPEDADDRESS }

See also [Compiler directives](#)

This compiler directive controls the type of pointer value the @ operator returns when applied to a variable reference.

Menu command: Options|Project|Compiler Options|Typed @ Operator

Syntax: {\$T+} or {\$T-}
 {\$TYPEDADDRESS ON} or {\$TYPEDADDRESS OFF}

Default: {\$T-}
 {\$TYPEDADDRESS OFF}

Scope: Local

Off { \$T- }, { \$TYPEDADDRESS OFF }

When Typed @ Operator is off, the result of the @ operator is an untyped pointer (Pointer) compatible with all other pointer types.

On { \$T+ }, { \$TYPEDADDRESS ON }

When Typed @ Operator is on, the result of the @ operator is ^T, where T is the type of variable reference. For example, @ applied to an integer variable always returns an integer pointer type.

If you apply @ to a procedure, function or method, the type of the resulting pointer is always Pointer, regardless of the state of this option.

See also
[@ operator](#)

Pentium-safe FDIV operations directive { \$U }, { \$SAFEDIVIDE }

Compiler directives

The \$U directive controls generation of floating-point code that guards against the flawed FDIV instruction exhibited by certain early Pentium processors.

Syntax: {\$U+} or {\$U-}
 {\$SAFEDIVIDE ON} or {\$SAFEDIVIDE OFF}

Default: {\$U-}
 {\$SAFEDIVIDE OFF}

Scope: Local

On { \$U+ }, { \$SAFEDIVIDE ON }

In the **{ \$U+ }** state, all floating-point divisions are performed using a run-time library routine. The first time the floating-point division routine is invoked, it checks whether the processor's FDIV instruction works correctly, and updates the TestFDIV variable (declared in the System unit) accordingly. For subsequent floating-point divide operations, the value stored in TestFDIV is used to determine what action to take.

The following table shows the possible values of TestFDIV:

Value	Meaning
-1	FDIV instruction has been tested and found to be flawed.
0	FDIV instruction has not yet been tested.
1	FDIV instruction has been tested and found to be correct.

For processors that do not exhibit the FDIV flaw, **{ \$U+ }** results in only a slight performance degradation. For a flawed Pentium processor, floating-point divide operations may take up to three times longer in the **{ \$U+ }** state, but they will always produce correct results.

Off { \$U- }, { \$SAFEDIVIDE OFF }

In the **{ \$U- }** state, floating-point divide operations are performed using in-line FDIV instructions. This results in optimum speed and code size, but may produce incorrect results on flawed Pentium processors. You should use the **{ \$U- }** only in cases where you are certain that the code is not running on a flawed Pentium processor.

Var-string checking directive { \$V }, { \$VARSTRINGCHECKS }

Compiler directives

This compiler directive controls type-checking on short strings passed as variable parameters.

Syntax: { \$V+ } or { \$V- }
 { \$VARSTRINGCHECKS ON } or { \$VARSTRINGCHECKS OFF }

Default: { \$V+ }
 { \$VARSTRINGCHECKS ON }

Scope: Local

On { \$V+ }, { \$VARSTRINGCHECKS ON }

In the { \$V+ } state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types.

Off { \$V- }, { \$VARSTRINGCHECKS OFF }

In the { \$V- } state, any short string-type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

Windows stack frame directive { \$W }, { \$STACKFRAMES }

[Compiler directives](#)

This compiler directive generates special prolog and epilog code for **far** procedures and functions, for programs that run in Windows 3.0 real mode.

Syntax: {\$W+} or {\$W-}
 {\$STACKFRAMES ON} or {\$STACKFRAMES OFF}

Default: {\$W+}
 {\$STACKFRAMES ON}

Scope: Local

On { \$W+ }, { \$STACKFRAMES ON }

When Windows Stack Frame is on, the compiler generates stack frames for procedures and functions, even when they are not needed.

Off { \$W- }, { \$STACKFRAMES OFF }

When off, the compiler generates stack frames only when needed.

Remarks

Some debugging tools require stack frames to be generated for all procedures and functions, but other than that, you should never need to use the **{ \$W+ }** state.

Extended syntax directive `{X}`, `{EXTENDED SYNTAX}`

[Compiler directives](#)

This compiler directive enables or disables C++Builder's extended syntax.

Syntax: `{X+}` or `{X-}`
`{EXTENDED SYNTAX ON}` or `{EXTENDED SYNTAX OFF}`

Default: `{X+}`
`{EXTENDED SYNTAX ON}`

Scope: Global

On `{X+}`, `{EXTENDED SYNTAX ON}`

When the Extended Syntax option is on, the Object Pascal syntax is extended so you can use user-defined function calls as statements (as if they were procedures). Extended syntax also allows you to use null-terminated strings.

Function calls can be used as statements; the result of a function call can be discarded. However, Extended Syntax does not apply to built-in functions (functions defined in the System Unit).

Extended Syntax also enables support for null-terminated strings by activating the special rules that apply to the built-in PChar type and zero-based character arrays.

Off `{X-}`, `{EXTENDED SYNTAX OFF}`

When Extended Syntax is off, attempts to use these extensions result in a compiler error.

Symbol information directive { \$Y }, { \$REFERENCEINFO }

See also [Compiler directives](#)

This compiler directive enables or disables generation of symbol reference information for debugging purposes.

Syntax: {\$Y+} or {\$Y-}
 {\$REFERENCEINFO ON} or {\$REFERENCEINFO OFF}

Default: {\$Y+}
 {\$REFERENCEINFO ON}

Scope: Global

Remarks

Symbol reference information consists of tables that provide the line numbers of all declarations of and references to symbols in a module.

On { \$Y+ }, { \$REFERENCEINFO ON }

When a program or unit is compiled with symbol information, the ObjectBrowser can display symbol definition and reference information in that module.

The symbol reference information for units is recorded in the .DCU file along with the unit's object code. Symbol reference information increases the size of the unit files, but does not affect the size or speed of the executable program.

The \$Y switch has no effect unless both the \$D and \$L switches are enabled.

Off { \$Y- }, { \$REFERENCEINFO OFF }

When off, disables generation of symbol reference information.

See also

[Debug and symbol information](#)

Include file directive { \$I filename }, { \$INCLUDE filename }

Compiler directives

Instructs the compiler to include the named file in the compilation.

Syntax: {\$I filename}
 {\$INCLUDE filename}

Scope: Local

Remarks

The default extension for filename is .PAS.

If filename does not specify a directory, C++Builder searches for the file

- First in the directory of the current source
- Then in the search path

The included file is inserted in the compiled text right after the {\$I filename} directive.

Note: An Include file cannot be specified in the middle of a statement part. All statements between the **begin** and **end** of a statement part must reside in the same source file.

Link object file directive { \$L filename }, { \$LINK filename }

[Compiler directives](#)

Instructs the compiler to link the named file with the program or unit being compiled.

Syntax: {\$L filename}
 {\$LINK filename}

Scope: Local

Remarks

The \$L directive is used to link in external routines written in other languages for procedures and functions declared to be **external**.

The named file must be an Intel relocatable object file (.OBJ file).

The default extension for filename is .OBJ.

If filename does not specify a directory, C++Builder searches

- First in the directory of the current source
- Then in the search path

Run-time type information { \$M }, { \$TYPEINFO }

[Compiler directives](#)

Controls generation of run-time type information.

Syntax: {\$M+} or {\$M-}
 {\$TYPEINFO ON} or {\$TYPEINFO OFF}

Default: {\$M-}
 {\$TYPEINFO OFF}

Scope: Local

Remarks

The **\$M** switch directive controls generation of run-time type information. When a class is declared in the **{ \$M+ }** state, or is derived from a class that was declared in the **{ \$M+ }** state, the compiler generates run-time type information for fields, methods, and properties that are declared in a published section. If a class is declared in the **{ \$M- }** state, and is not derived from a class that was declared in the **{ \$M+ }** state, published sections are not allowed in the class.

Note The TPersistent class defined in the C++Builder Visual Component Library (VCL) was declared in the **{ \$M+ }** state, so any class derived from TPersistent is allowed to contain published sections. VCL uses the run-time type information generated for published sections to access the values of a component's properties when saving and loading form files. Furthermore, the IDE uses a component's run-time type information to determine the list of properties to show in the Object Inspector.

There is seldom, if ever, any need for an application to directly use the **\$M** compiler switch.

Memory allocation size directives `{ $M }`, `{ $MAXSTACKSIZE }`, `{ $MINSTACKSIZE }`

[Compiler directives](#)

Specifies a program's stack allocation parameters.

Syntax: `{ $M minstacksize, maxstacksize }`
`{ $MINSTACKSIZE number }`
`{ $MAXSTACKSIZE number }`

Default: `{ $M 16384, 1048576 }`

Scope: Global

Remarks

The **\$M** directive specifies an application's stack allocation parameters. `minstacksize` must be an integer number between 1024 and 2147483647 which specifies the minimum size of an application's stack, and `maxstacksize` must be an integer number between `minstacksize` and 2147483647 which specifies the maximum size of an application's stack.

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.

An application's stack is never allowed to grow larger than the maximum stack size. Any attempt to grow the stack beyond the maximum stack size causes an `EStackOverflow` exception to be raised.

The **\$MINSTACKSIZE** and **\$MAXSTACKSIZE** directives allow the minimum and maximum stack sizes to be specified separately.

Note The memory allocation directives are meaningful only in a program. They should not be used in a library or a unit.

Description directive { \$D text }, { \$DESCRIPTION text }

Compiler directives

Inserts the specified text into the module description entry in the header of an EXE file or DLL.

Syntax: {\$D text}
 {\$DESCRIPTION text}

Scope: Global

Remarks

Only one description directive can appear in a program or DLL source file. Do not use \$D in unit source files.

Resource file directive { \$R filename }, { \$RESOURCE filename }

[Compiler directives](#)

Specifies the name of the resource file to be included in an application or library.

Syntax: {\$R filename}
 {\$RESOURCE filename}

Scope: Local

Remarks

The default extension for `filename` is `.RES`. It must be a Windows resource file.

If `filename` does not specify a directory, the compiler searches for the file

- First in the directory of the current source
- Then in the search path

When used in a unit, the resource file name is simply recorded in the resulting unit file; no checks are made to ensure that the file exists at compile time.

When an application or library is linked, the resource files specified in all units and in the program or library itself are processed and each resource in each resource file is copied to the `.EXE` or `.DLL` file being produced.

Note: This directive allows multiple `.RES` files per unit. There is no compile-time confirmation of the contents of a `.RES` file, or whether it is a valid `.RES` file (whether it exists). Files listed with the `$R` directive must be present at link time, or you will receive the error message "File not found (<filename>.RES)."

Minimum enumeration size directive { \$Z }, { \$MINENUMSIZE }

[Compiler directives](#)

This directive controls the minimum storage size of enumerated types.

Syntax: {\$Z1} or {\$Z2} or {\$Z4}
 {\$MINENUMSIZE 1} or {\$MINENUMSIZE 2} or {\$MINENUMSIZE 4}

Default: {\$Z1}
 {\$MINENUMSIZE 4}

Scope: Local

Remarks

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values, and if 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. Finally, if an enumerated type is declared in the **{ \$Z4 }** state, it is stored as an unsigned double-word.

The **{ \$Z2 }** and **{ \$Z4 }** states are useful for interfacing with C and C++ libraries, which usually represent enumerated types as words or double-words.

Warnings directive { \$WARNINGS }

[See also](#) [Compiler directives](#)

This compiler directive controls whether the generation of compiler warnings.

Syntax: { \$WARNINGS ON } or { \$WARNINGS OFF }

Default: { \$WARNINGS OFF }

Scope: Local

On { \$WARNINGS ON }

When Warnings is on, the compiler generates warning messages in the Message Window when it detects uninitialized variables, missing function results, construction of abstract objects, and so on.

Off { \$WARNINGS OFF }

When off, the compiler does not generate warning messages.

See also

[Hints directive](#)

Hints directive { \$HINTS }

[See also](#) [Example](#) [Compiler directives](#)

The \$HINTS directive controls whether the compiler generates hint messages at compile time.

Syntax: {\$HINTS ON} or {\$HINTS OFF}

Default: {\$HINTS OFF}

Scope: Local

On { \$HINTS ON }

When Hints is on, the compiler issues hint messages in the Message Window when it detects unused variables, unused assignments, **for** or **while** loops that never execute, and so on.

Off { \$HINTS OFF }

When off, the compiler does not generate hint messages.

Remarks

By placing code between {\$HINTS OFF} and {\$HINTS ON} directives, you can selectively turn off hints that you don't care about.

Example

{ The following example shows how to prevent the compiler from generating hints on an unused variable. }

```
{ $HINTS OFF }  
procedure Test;  
var  
    I: Integer;  
begin  
end;  
{ $HINTS ON }
```

See also

[Warnings directive](#)

Code-image base directive { **\$IMAGEBASE** address }

Compiler directives

The **\$IMAGEBASE** directive specifies the default load address for an application or DLL.

Syntax: {**\$IMAGEBASE** number}
Default: {**\$IMAGEBASE** \$00400000}
Scope: Global

Remarks

The number argument must be a 32-bit integer value that specifies image base address. The number argument must be greater than or equal to \$00010000, and the lower 16 bit of the argument are ignored and should be zero.

When a module (application or DLL) is loaded into the address space of a process, Windows will attempt to place the module at its default image base address. If that does not succeed, that is if the given address range is already reserved, the module is relocated to an address assigned by Windows.

There is seldom, if ever, any reason to change the image base address of an application. For a DLL, however, it is recommended that you use the **\$IMAGEBASE** directive to specify a non-default image base address, since the default image base address of \$00400000 will almost certainly never be available. The recommended address range of DLL images is \$40000000 to \$7FFFFFFF. Addresses in this range are always available to a process in both Windows NT and Windows 95.

When Windows succeeds in loading a DLL at its image base address, the load time of the DLL is decreased because relocation fixups do not have to be applied. Furthermore, when the given address range is available in multiple processes that use the DLL, code portions of the DLL's image can be shared among the processes, thus reducing load time and memory consumption.

Long strings directive { \$H }, { \$LONGSTRINGS }

[See also](#) [Compiler directives](#)

The **\$H** directive controls the meaning of the reserved word **string** used alone in a type declaration.

Syntax: {\$H+} or {\$H-}
 {\$LONGSTRINGS ON} or {\$LONGSTRINGS OFF}

Default: {\$H+}
 {\$LONGSTRINGS ON}

Scope: Global

Remarks

The generic type **string** can represent either a long, dynamically-allocated string (the fundamental type `AnsiString`) or a short, statically-allocated string (the fundamental type `ShortString`).

On { \$H+ }, { \$LONGSTRINGS ON }

By default {\$H+}, C++Builder defines the generic string type to be the long `AnsiString`. All components in the Visual Component Library (VCL) are compiled in this state. If you write components, they should also use long strings, as should any code that receives data from VCL string-type properties.

Off { \$H- }, { \$LONGSTRINGS OFF }

The {\$H-} state is mostly useful for using code from versions of Object Pascal that used short strings by default. You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to **string**[255] or `ShortString`, which are unambiguous and independent of the **\$H** setting.

See also

[Long string types](#)

[Short string types](#)

Optimization directive { \$O }, { \$OPTIMIZATION }

Compiler directives

Syntax: {\$O+} or {\$O-}
 {\$OPTIMIZATION ON} or {\$OPTIMIZATION OFF}

Default: {\$O+}
 {\$OPTIMIZATION ON}

Scope Local

The **\$O** directive controls code optimization. In the **{\$O+}** state, the compiler performs a number of code optimizations, such as placing variables in CPU registers, eliminating common subexpressions, and generating induction variables. In the **{\$O-}** state, all such optimizations are disabled.

Other than for certain debugging situations, you should never have a need to turn optimizations off. All optimizations performed by C++Builder's Object Pascal compiler are guaranteed not to alter the meaning of a program. In other words, C++Builder performs no "unsafe" optimizations that require special awareness by the programmer.

Language definition

[Language reference](#)

The topics listed below are the elements of the formal Object Pascal language definition.

Some of these topics use syntax diagrams to illustrate aspects of the Object Pascal language. If you do not know how to read a syntax diagram, see the topic [How to read a syntax diagram](#).

You can access this material from the from the Help Contents screen, from the Help search engine, or directly from the Code Editor by pressing Ctrl+F1.

[Arrays](#)

[Blocks](#)

[Character strings](#)

[Comments](#)

[Compiler directives](#)

[Constant declarations](#)

[DLLs](#)

[Exception handling](#)

[Expressions](#)

[Functions](#)

[Identifiers](#)

[Labels](#)

[Loops](#)

[Methods](#)

[Numbers](#)

[Operators](#)

[Procedures](#)

[Reserved words](#)

[Scope](#)

[Special symbols](#)

[Statements](#)

[Strings](#)

[Tokens](#)

[Typed constants](#)

[Type declarations](#)

[Variables](#)

[Units](#)

Language reference

[Language definition](#)

You can access this material from the Help menu, from the Help Contents screen, or directly from the Code Editor by pressing Ctrl+F1.

[Compiler directives](#)

[Components](#)

[Conditional directives and symbols](#)

[Procedures and Functions \(categorical\)](#)

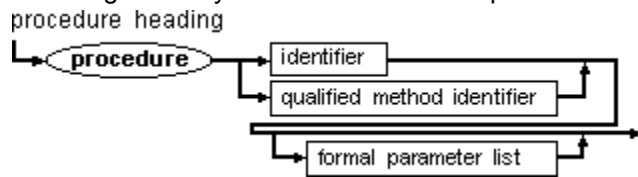
[Reserved words](#)

[Standard directives](#)

[Units](#)

How to read a syntax diagram

Knowing how to read a syntax diagram is a basic skill necessary for learning the Object Pascal language. Throughout this Help system you will encounter syntax diagrams. They represent the proper ordering of the syntax elements for that part of the language.



To read a syntax diagram, follow the arrows. Frequently, more than one path is possible and all are legal.

Actual terms that are used in your code are shown in **bold** type in the diagrams.

The shapes in the syntax diagram represent specific syntax elements. They are:

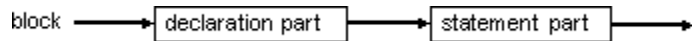
Shape	Represents
Box	Constructions
Circle	Reserved words, operators, and punctuation

Blocks

[See also](#)

A block is made up of statements.

Blocks are part of a procedure declarations, function declarations, method declarations, or a program or unit.



Declaration part

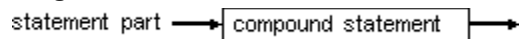
The declaration part of a block can contain any of the following:

- [Labels](#)
- [Constants](#)
- [Types](#)
- [Variables](#)
- [Procedures](#)
- [Functions](#)
- [Exports clause](#)

All identifiers and labels that you declare in a block are local in [scope](#) to that block.

Statement part

The statement part of a block is a [compound statement](#): that is, it is delimited by the reserved words **begin** and **end** and contains one or more statements.



See also

[Begin..end](#)

[Block scope](#)

[Statements](#)

Constant declarations

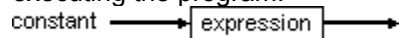
[See also](#)

A constant is an identifier that represents a value that cannot change. The scope of a constant is only within the block containing its declaration. A constant identifier cannot be included within its own declaration.



Constants are declared with the reserved word const.

Object Pascal lets you use constant expressions, which can be evaluated by the compiler without actually executing the program.



Since C++Builder has to completely evaluate a constant expression at compile time, the following constructs are not allowed in constant expressions:

- References to variables and typed constants (except in constant address expressions)
- The @ operator (except in constant address expressions)
- Function calls (except for the following)

Abs	Odd
Addr	Ord
Chr	Pred
Hi	Round
High	SizeOf
Length	Succ
Lo	Swap
Low	Trunc

The following binary arithmetic and Boolean operators can also be used in constant expressions:

+	shr
-	shl
/	and
div	or
mod	xor
=	

See also

Scope

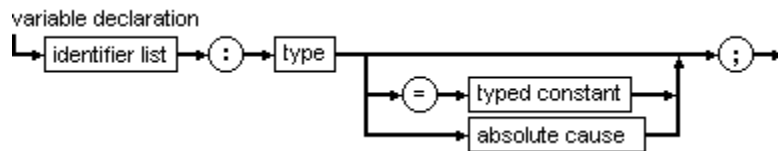
Typed constants

Variables

See also [Example](#)

A variable is an identifier that represents a value that can change. You can declare variables within the variable declaration part of a unit, procedure, function, or program.

In a variable declaration, you must declare an identifier and its associated type.



The type given for the variable(s) can be a type identifier previously declared in one of the following:

- A **type** declaration part in the same block
- An enclosing block
- A unit

Variables can also declare new types.

The scope of a variable identifier is within the block in which the declaration occurs. The variable can be referred to throughout the block, unless the identifier is redeclared in an enclosed block.

Redeclaration creates a new variable using the same identifier, without affecting the value of the original variable.

See also

[Global and local variables](#)

[Scope](#)

[Unit](#)

[Var \(reserved word\)](#)

[Variable reference](#)

[Variable typecasting](#)

[Initialized variables](#)

Examples

var

```
X, Y, Z: Double;  
I, J, K: Integer;  
Digit: 0..9;  
C: Color;  
Done, Error: Boolean;  
Operator: (Plus, Minus, Times);  
Hue1, Hue2: set of Color;  
Today: Date;  
Matrix: array[1..10, 1..10] of Double;
```

Global and local variables

See also [Variables](#)

Global variables are declared outside procedures and functions. Global variables are available to all

- [Procedures](#)
- [Functions](#)
- [Methods](#)

Local variables are declared within procedures, functions, and methods. They are available only within the enclosing [block](#) and are destroyed when the procedure or function returns to the caller.

Local variables and the stack

Variables declared within procedures and functions are called local variables, and reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables on the stack. On exit, the local variables are disposed of.

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 (1M) respectively. An application is guaranteed to always 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 an application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated as needed 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.

See also

Scope

Initialized variables

See also [Variables](#)

When a variable declaration declares a single global variable, the declaration can optionally specify an initial value for the variable. If a global variable declaration does not explicitly specify an initial value, the memory occupied by the variable will initially be set to zero.

It is not possible to specify the initial value for a local variable, and upon entry to a procedure or function, all local variables have undefined values.

See also

[Typed constants](#)

[Global and local variables](#)

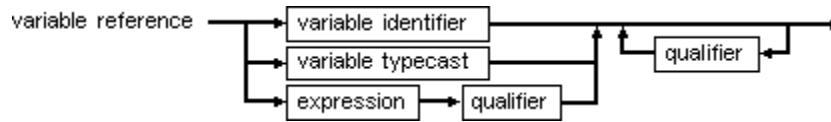
Variable references

[See also](#)

A variable reference signifies one of the following:

- A variable
- A component of a structured- or string-type variable
- A dynamic variable pointed to by a pointer-type variable

The structure for a variable reference is:



The syntax for a variable reference allows an expression that computes a pointer-type value. The expression must be followed by a qualifier that dereferences the pointer value (or indexes the pointer value if the extended syntax is enabled with the `{$X+}` directive) to produce an actual variable reference.

See also

[Pointer-type variables](#)

[Qualifiers](#)

[String-type variables](#)

[Structured-typed variables](#)

[Variables](#)

[Variable typecasting](#)

Qualifiers

See also [Variable references](#)

Qualifiers modify the meaning of a variable reference. A variable can contain zero or more qualifiers.



An array identifier that references the whole array has no qualifier.

An array identifier followed by an index represents a specific component of the array.

With a component that is a record or object, you can follow the index with a field designator which represents a specific field within a specific array component.

You can follow the field designator in a pointer field with the pointer symbol (^) to differentiate between the pointer field and the dynamic variable to which it points.

If the variable being pointed to is an array, you can add indexes to denote components of this array.

The pointer symbol is optional when dereferencing a structured type.

See also

[Array types](#)

[Field and object component designators](#)

[Indexes](#)

[Pointers and dynamic variables](#)

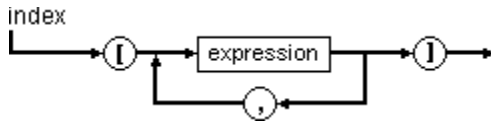
[Pointer types](#)

[Variable references](#)

Indexes

See also [Variable references](#)

Indexes provide a means for accessing a specific element of an array, a string, or a string list.



Array indexes

The index of an array lets you access a specific component of an array.

The index expression selects components in each corresponding dimension of the array. The following restrictions apply to array indexes:

- The number of expressions cannot exceed the number of index types in the array declaration.
- The type of each expression must be assignment-compatible with the corresponding index type.

String indexes

You can index a short string variable with a single index expression, whose value must be in the range 0..N, where N is the declared maximum length of the short string. The type of a character accessed through indexing of a short string is Char. The index of the first character in a string is 1. The element at index 0 contains the dynamic length of the string.

You can index a non-empty long string variable with a single index expression, whose value must be in the range 1..N, where N is the dynamic length of the long string. The type of a character accessed through indexing of a long string is Char. The index of the first character in a long string is 1.

A value of type PChar, PAnsiChar, or PWideChar can be indexed with a single index expression of type *Integer*. The index expression specifies an offset (number of characters or wide characters) to add to the character pointer before it is dereferenced to produce a Char, AnsiChar, or WideChar type variable reference.

To determine the length of a string,

- Use the [Length](#) function.

String list indexes

The index of a string list lets you access a particular string in the list.

The string list has an indexed property called [Strings](#), which you can treat like an array of strings.

Since the Strings property is the most common part of a string list to access, Strings is the default property of the list, meaning that you can omit the Strings identifier and just treat the string list itself as an indexed array of strings.

To access a particular string in a string list, refer to it by its index. The string numbers are zero-based, so if a list has three strings in it, the indexes cover the range 0..2.

To determine the maximum index, check the [Count](#) property. If you try to access a string outside the range of valid indexes, the string list raises an exception.

See also

[Array types](#)

[Qualifiers](#)

[String types](#)

[Working with string lists](#)

Example

The following example accesses a cell of an array.

```
Matrix[I, J];
```

The following examples do exactly the same thing, setting the first line of text in a memo field:

```
Memo1.Lines.Strings[0] := 'This is the first line.';  
Memo1.Lines[0] := 'This is the first line.';
```

Field and object designators

[See also](#)

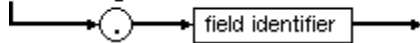
[Example](#)

[Variable references](#)

Field designators

A field designator provides access to a specific field in a record.

field designator



In a statement within a **with** statement, a field designator does not have to be preceded by a variable reference to its containing record.

Object component designators

The object component designator provides access to a specific component of an object. A component designator that designates a method is called a method designator.

The instance and the period can be omitted in the following cases:

- When referencing components using the **with** statement
- Within a method block because the effect is the same as if `Self` and a period were written before the component reference

See also

[Class types](#)

[Record types](#)

[With statement](#)

Example

The following examples access fields within a record.

Today.Year

Results[1].Count

Results[1].When.Month

Pointers and dynamic variables

[See also](#) [Example](#)

Pointer variables contain a value of **nil** or the address of a dynamic variable.

The dynamic variable pointed to by a pointer variable is referenced by writing the pointer symbol (^) after the pointer variable.

You can create dynamic variables and their pointer values using the [New](#) and [GetMem](#) procedures.

You can use the [@](#) (address-of) operator and the function [Addr](#) to create pointer values that are treated as pointers to dynamic variables.

nil does not point to any variable. The results are undefined if you access a dynamic variable when the value of the pointer is **nil** or undefined.

See also

[Pointer types](#)

[Variables](#)

Example

The following examples are references to dynamic variables.

`P1^`

`P1^.Siblings^`

`Results[1].Data^`

Variable typecasting

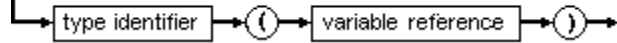
[See also](#)

[Example](#)

[Variables](#)

Variable typecasting changes the variable reference of one type into a variable reference of another type.

variable typecast



When a variable typecast is applied to a variable reference, the variable reference is treated as an instance of the type specified by the type identifier. The size of the variable must be the same as the size of the type denoted by the type identifier. If a typecast is performed using the as operator, the validity of the cast is checked, and an exception is raised if the variable is not assignment-compatible with the type to which it is cast. No checks are performed if the typecast is written using the name of the type followed by a variable name in parentheses.

A variable typecast can be followed by one or more qualifiers, as allowed by the specific type.

Object Pascal supports variable typecasts involving procedural types.

See also

Qualifiers

Value typecasts

Example

Given the following declarations:

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

You can construct 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 P to F }
N := F(N);         { Call function via F }
N := Func(P)(N);  { Call function via P }
```

Identifiers

[Example](#) [Language definition](#)

Identifiers are descriptive names you assign to any element of an Object Pascal program.

Constants

Fields in records

Functions

Labels

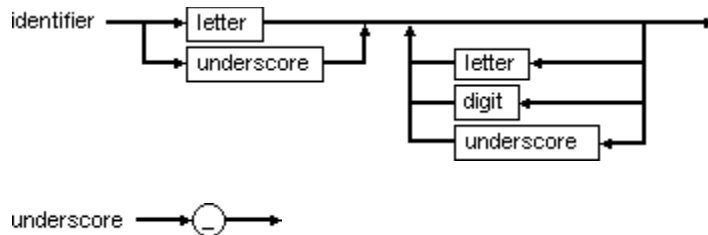
Procedures

Programs

Types

Units

Variables



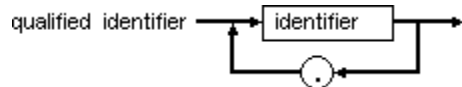
Identifiers are not case-sensitive.

The following restrictions apply to identifier names:

- Identifiers can be of any length, but only the first 63 characters are significant.
- The first character of an identifier must be a letter or an underscore (_).
- The characters that follow the first one must be letters, digits, or underscores.
- No spaces are allowed in an identifier.

Qualified identifiers

Qualified identifiers are helpful in preventing name conflicts when several instances of the same identifier exist. You can qualify the identifier with another identifier in order to select a specific instance.



Examples

The following items are regular identifiers.

```
TextFile  
Exit  
Real2String
```

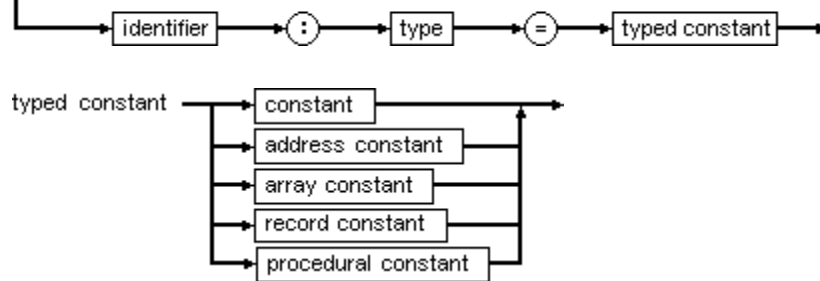
The following items are qualified identifiers.

```
System.MemAvail (* unit = System, identifier = MemAvail *)  
System.CloseFile;
```

Typed constants

[See also](#) [Example](#)

The declaration of a typed constant corresponds to the declaration of a read-only variable. Typed constants can be used exactly like variables of the same type, except that they cannot be modified.



In addition to a normal constant expression, the value of a typed constant can be specified with a constant address expression.

There are five categories of typed constants:

Pointer-type constants

Procedural-type constants

Simple-type constants

String-type constants

Structured-type constants

Note The **\$J** compiler directive allows the declaration of typed constants that can be modified. Typed constants declared in the default **{**\$J**-}** state are read-only and cannot be modified.

See also

[Constant declarations](#)

[Initialized variables](#)

Examples

(* Typed Constant Declarations *)

type

Point = **record** X, Y: real **end**;

const

Minimum: Integer = 0;

Maximum: Integer = 9999;

Factorial: **array**[1..7] **of** Integer = (1, 2, 6, 24, 120, 720, 5040);

HexDigits: **set of** Char = ['0'..'9', 'A'..'Z', 'a'..'z'];

Origin: Point = (X: 0.0; Y: 0.0);

Structured-type constants

See also [Typed constants](#)

The declaration of a structured-type constant specifies the value of each of the structure's components.

Object Pascal supports the declaration of the following type constants:

array

record

set

pointer

File-type constants and constants of **array**, and **record** types that contain **file**-type components are not allowed.

See also

[Pointer-type constants](#)

[Procedural-type constants](#)

[Simple-type constants](#)

[String-type constants](#)

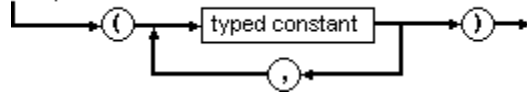
Array-type constants

[Examples](#)

[Structured-type constants](#)

An array-type constant declares an array with preinitialized elements.

array constant



The declaration of an array-type constant specifies the values of the components. The component type of an array-type constant can be any type except a file type.

Character arrays

Packed string-type constants (character arrays) can be specified either as single characters or as strings.

Zero-based character arrays

The index of the first element of a zero-based character array is 0, and that of the last element is a positive nonzero integer.

A zero-based character array can be initialized with a string that is shorter than the declared length of the array. When the string is shorter than the array's length, the remaining characters are set to NULL (#0) and the array will effectively contain a null-terminated string.

Multidimensional array constants

Multidimensional array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas.

The innermost constants correspond to the rightmost dimensions.

Examples

The following example constructs the array-type constant StatStr.

type

```
TStatus = (Active, Passive, Waiting);  
TStatusMap = array[TStatus] of string;
```

const

```
StatStr: TStatusMap = ('Active', 'Passive', 'Waiting');
```

These are the components of StatStr:

```
StatStr[Active] = 'Active'  
StatStr[Passive] = 'Passive'  
StatStr[Waiting] = 'Waiting' }
```

The following example declares an initialized multidimensional array Maze.

type

```
TCube = array[0..1, 0..1, 0..1] of Integer;
```

const

```
Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

These are the values for the array Maze:

```
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
```

Pointer-type constants

[See also](#) [Example](#) [Typed constants](#)

The declaration of a pointer-type constant typically uses a constant address expression to specify the pointer value.

A typed constant of type PChar can be initialized with a string constant.

See also

[PChar](#)

[Pointer types](#)

Examples

The following example declares pointer-type constants.

type

```
TDirection = (Left, Right, Up, Down);
PNode = ^Node;
TNode = record
  Next: PNode;
  Symbol: string;
  Value: TDirection;
end;
```

const

```
N1: TNode = (Next: nil; Symbol: 'DOWN'; Value: Down);
N2: TNode = (Next: @N1; Symbol: 'UP'; Value: Up);
N3: TNode = (Next: @N2; Symbol: 'RIGHT'; Value: Right);
N4: TNode = (Next: @N3; Symbol: 'LEFT'; Value: Left);
```

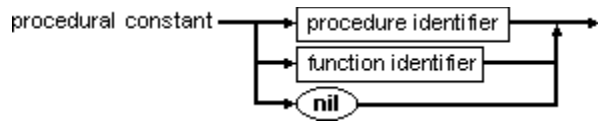
var

```
DirectionTable: PNode = @N4;
```

Procedural-type constants

[See also](#) [Example](#) [Typed constants](#)

A procedural-type constant lets you preinitialize procedural types.



A procedural-type constant must specify the identifier of a procedure or function that is assignment compatible with the type of the constant, or it must specify the value **nil**.

See also

[Procedural types](#)

[Typed constants](#)

Example

The following example assigns a procedure to a type constant.

type

```
TErrorProc = procedure(ErrorCode: Integer);
```

procedure DefaultError(ErrorCode: Integer);

begin

```
  WriteLn('Error ', ErrorCode, '.');
```

end;

const

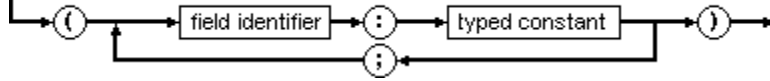
```
  ErrorHandler: TErrorProc = DefaultError;
```

Record-type constants

[See also](#) [Example](#) [Structured-type constants](#)

The declaration of a record-type constant specifies the identifier and value of each field, enclosed in parentheses and separated by commas.

record constant



The fields must be specified in the same order as they appear in the definition of the record type.

- If a record contains fields of **file** types, constants of that record type cannot be declared.
- If a record contains a variant, only fields of the selected variant can be specified.
- If the variant contains a tag field, its value must be specified.

See also

Records

Typed constants

Examples

The following example declares the record-type constant TPoint.

type

```
TPoint = record
  X, Y: Single;
end;
TVector = array[0..1] of Point;
TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jly, Aug, Sep, Oct, Nov, Dec);
TDate = record
  D: 1..31;
  M: Month;
  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);
```

Set-type constants

[See also](#)

[Example](#)

[Structured-type constants](#)

A set-type constant lets you preinitialize the elements of a set constant.

The declaration of a set-type constant specifies the value of the set using a constant expression.

See also

[Sets](#)

[Set types](#)

[Typed Constants](#)

Examples

The following example declares a set-type constants for Digits and Letters.

type

```
Digits = set of 0..9;  
Letters = set of 'A'..'Z';
```

const

```
EvenDigits: Digits = [0, 2, 4, 6, 8];  
Vowels: Letters = ['A', 'E', 'I', 'O', 'U', 'Y'];  
HexDigits: set of '0'..'z' = ['0'..'9', 'A'..'F', 'a'..'f'];
```

Simple-type constants

[See also](#)

[Example](#)

[Typed constants](#)

The declaration of a simple-type constant specifies the value of the constant.

You can specify the value of a typed constant using a constant address expression.

Because a typed constant is actually a variable with a constant value, it cannot be used in the declaration of other constant types.

See also

Types

Typed constants

Examples

The following example declares simple constants.

const

```
Maximum: Integer = 9999;
```

```
Factor: Real = -0.1;
```

```
Breakchar: Char = #3;
```

String-type constants

[See also](#) [Examples](#) [Typed constants](#)

The declaration of a typed constant of a string type simply specifies the string constant:

To declare a short string typed constant, include a length specifier in the declaration:

See also

[String types](#)

[Typed constants](#)

String-type constant examples

The following declares long-string-type constants.

const

```
Heading: string = 'Section';  
NewLine: string = #13#10;  
TrueStr: string = 'Yes';  
FalseStr: string = 'No';
```

The following declares a short-string-type constant.

const

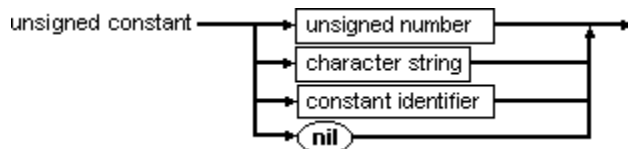
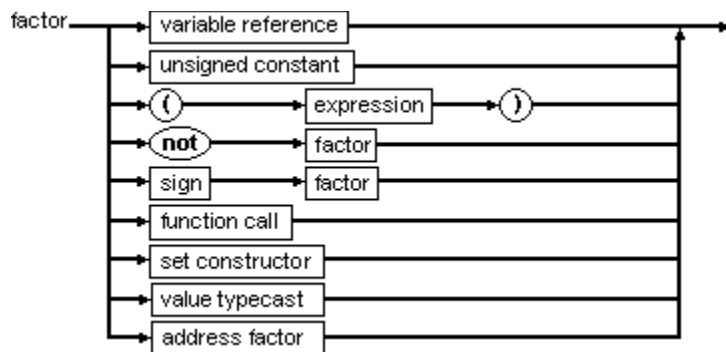
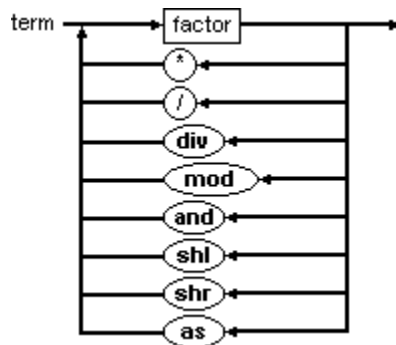
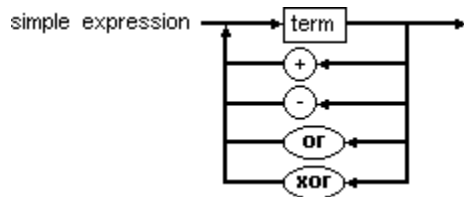
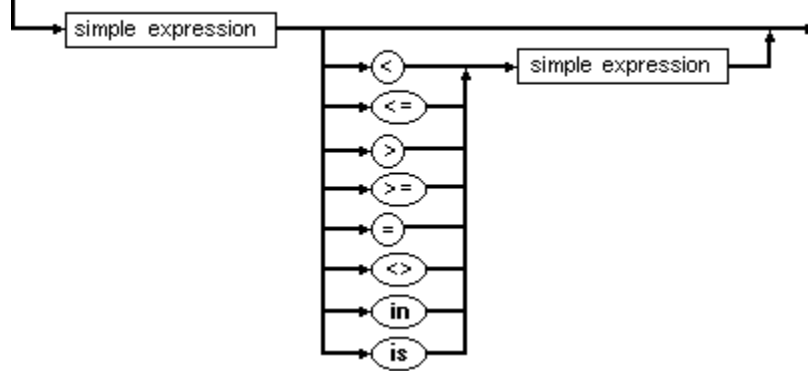
```
ShortStr: string[5] = 'Short';.
```

Expressions

See also

Expressions are a combination of operators and operands that evaluate to a single resulting value.

expression



These are the operands:

Constants

Function calls

Procedure statements

Set constructors

Variables

Subexpressions can be enclosed in parentheses to change the order of precedence.

See also

[@ operator](#)

[Blocks](#)

[Comments](#)

[Constant declarations](#)

[Function calls](#)

[Precedence of operators](#)

[Set types](#)

[Statements](#)

[Value typecasts](#)

[Variable reference](#)

Function calls

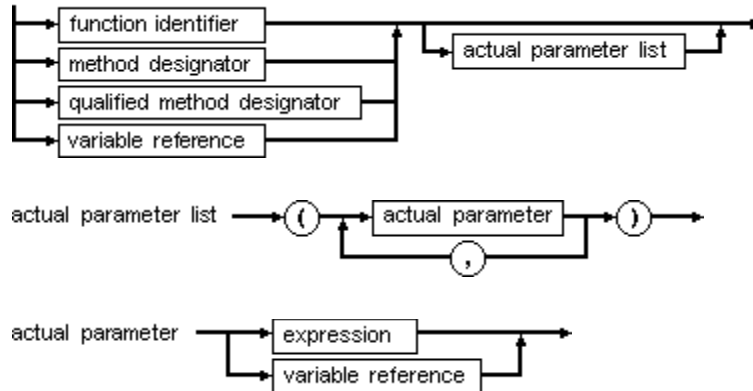
[See also](#)

A function call activates a function specified by one of the following:

- A function identifier
- A method designator
- A qualified method designator
- A procedural-type variable reference

If the corresponding function declaration contains a list of formal parameters, the function call must include a list of actual parameters. Each parameter takes the place of the corresponding formal parameter according to parameter rules.

function call



Object Pascal allows the result of a function call to be discarded, in essence treating the function call as a procedure statement.

See also

[Method activation](#)

[Qualified-method activations](#)

[Parameters](#)

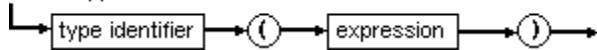
[Procedural types](#)

Value typecasting

[See also](#) [Example](#)

Value typecasting is the process of changing the type of an expression to another type.

value typecast



The expression type and the specified type must both be one of the following:

- An ordinal type
- A pointer type

For ordinal types, the resulting value is obtained by converting the expression, which may involve truncation or extension of the original value if the size of the specified type is different from that of the expression. In cases where the value is extended, the sign of the value is always preserved.

Value typecasts operate on values and cannot be followed by a qualifier.

Example

The following statements are examples of value typecasting.

```
Integer('A')
```

```
Char(48)
```

```
Boolean(0)
```

```
Color(2)
```

```
Longint(@Buffer)
```

See also

[Variable typecasting](#)

Using procedural types in expressions

[See also](#)

[Example](#)

Using a procedural variable in a statement of an expression calls the procedure or function stored in the variable. However, when the compiler sees a procedural variable on the left side of an assignment statement, it knows that the right side has to represent a procedural value. Unfortunately, there are situations where the compiler cannot determine the action you want from the context.

Procedural types and the @ operator

When you apply the address (@) operator to a procedure or function identifier, the argument is converted into a pointer, and the compiler is prevented from calling the procedure.

The @ operator is often used when assigning an untyped pointer value to a procedural variable.

To get the memory address of a procedural variable rather than the address stored in it, use a double address (@@) operator.

Example

type

```
  IntFunc = function: Integer;
```

var

```
  F: IntFunc;
```

```
  N: Integer;
```

function ReadInt: Integer;

var

```
  I: Integer;
```

begin

```
  Read(I);
```

```
  ReadInt := I;
```

end;

begin

```
  F := ReadInt;    { Assign procedural value }
```

```
  N := ReadInt;    { Assign function result }
```

end.

See also

[@ operator](#)

[Procedural types](#)

Special symbols

[See also](#)

Special symbols are characters from the ASCII character set that have predefined meanings. Therefore, you can use them in your programs only as they are defined by the Object Pascal language.

The following single characters are special symbols:

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

The following character pairs are also special symbols:

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

Some of the special symbols are operators.

Certain special symbols have a character pair that performs the same function.

Character	Equivalent character pair
[(.
]).)
{	(*
}	*)

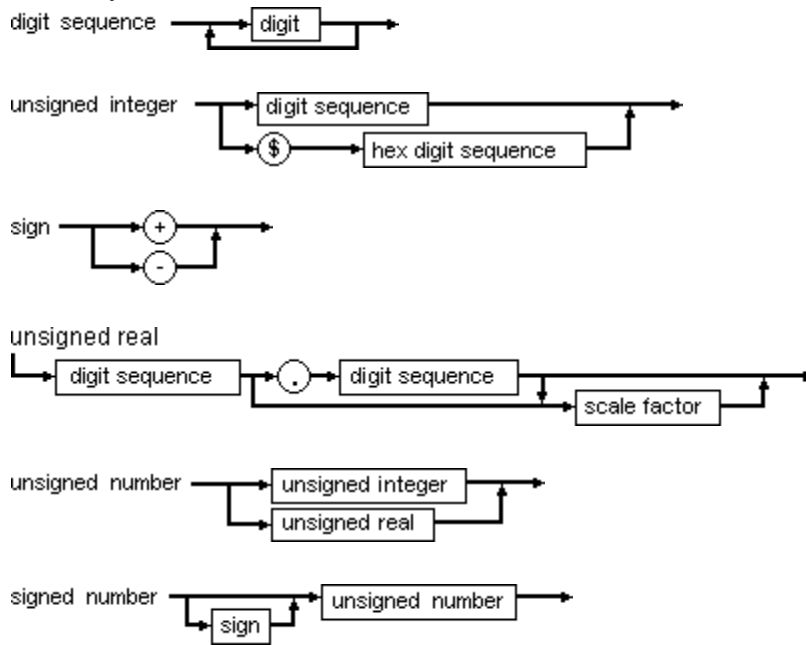
Object Pascal uses the following subsets of the ASCII character set:

- Letters -- the English alphabet, A through Z and a through z
- letter
-
- Digits -- the Arabic numerals 0 through 9
- digit
-
- Hex digits -- the Arabic numerals 0 through 9, the letters A through F, and the letters a through f
- hex digit
-
- Blanks -- the space character (ASCII 32) and all ASCII control characters (ASCII 0 through 31), including the end-of-line or return character (ASCII 13)

See also
Comments
Operators

Numbers

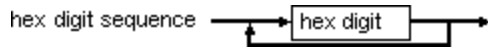
Ordinary decimal notation is used for numbers that are constants of integer and real types.



Numbers with decimals or exponents are real-type constants. Other decimal numbers denote integer-type constants; they must be between -2,147,483,648 and 2,147,483,647.

Hexadecimal numbers

Hexadecimal integer constants use a dollar sign (\$) as a prefix.



When hexadecimal numbers are used as integer-type constants; they must be between \$00000000 and \$FFFFFFFF. The resulting value's sign is implied by the hexadecimal notation.

Engineering notation

Engineering notation (E or e, followed by an exponent) is read as "times 10 to the power of" in real types. For example:

7E-2 means 7×10^{-2}

12.25e+6 or 12.25e6 both mean $12.25 \times 10^{+6}$.



Character strings

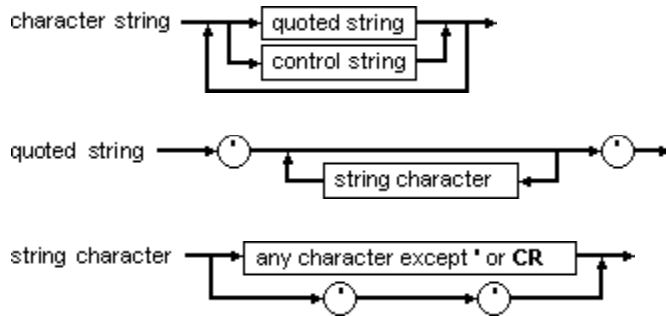
Example

A character string is a sequence of zero or more characters from the extended ASCII character set, written on one line in the program and enclosed by apostrophes.

A character string with nothing between the apostrophes is a null string.

Two sequential apostrophes in a character string represents a single apostrophe.

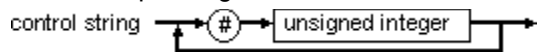
The length attribute of a character string is the actual number of characters within the apostrophes. The maximum length for a string is 255 characters.



Control characters

As an extension to standard Object Pascal, C++Builder allows the use of extended characters within character strings.

The # character followed by an unsigned integer constant between 0 and 255 represents a character of the corresponding ASCII value.



There must be no separators between the # character and the integer constant.

If several control characters are part of a character string, there must be no separators between them.

String compatibility

- A character string of length 0 (the null string) is compatible only with string types.
- A character string of length 1 is compatible with any Char and **string** type.
- A character string of length N, where N is greater than or equal to 2, is compatible with the following:
 - Any string type
 - Packed arrays of N characters
 - The PChar type when extended syntax is enabled with the {SX+} compiler directive

Examples

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

Comments

[See also](#)

[Example](#)

If you want to clarify the purpose of a block of code, you can include explanatory comments by enclosing the text in braces `{ }` or asterisks/parentheses `(* *)`.

The text between the comment delimiters is ignored by the compiler.

You cannot include an end-of-comment delimiter (`}` or `*)` within the comment text because the compiler recognizes that as closing the comment.

A comment containing a dollar sign (\$) immediately after the opening `{` or `(*` is a compiler directive. A mnemonic of the compiler command follows the \$ character.

You can also create a single-line comment by putting two slashes (`//`) in front of the comment text. The compiler then ignores everything until the end of the line.

See also

Blocks

Examples

```
{ Any text not containing right brace }  
(* Any text not containing asterisk/right parenthesis *)  
// Any text from a double-slash to the end of the line
```


Tokens

[See also](#)

Tokens are the smallest meaningful units of text in a Object Pascal program. Tokens include

- [Character strings](#)
- [Identifiers](#)
- [Labels](#)
- [Numbers](#)
- [Reserved words](#)
- [Special symbols](#)

When you use two consecutive tokens in a program, you need to include a [separator](#) between them if either token is a reserved word, an identifier, a label, or a number. You cannot use separators as part of tokens except in string constants.

See also

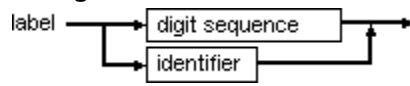
[Comments](#)

[Standard directives](#)

Labels

[See also](#)

A label is a digit sequence in the range 0 to 9999 (leading zeros are not significant) that marks the target of a **goto** statement.



As an extension to Standard Pascal, Object Pascal also allows identifiers to function as labels.

See also

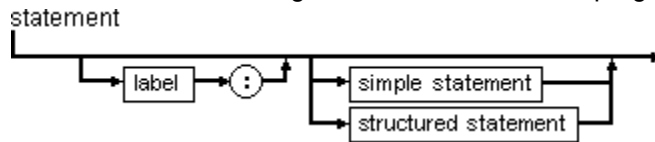
Blocks

Label (reserved word)

Statements

[See also](#)

Statements describe algorithmic actions that the program can execute.

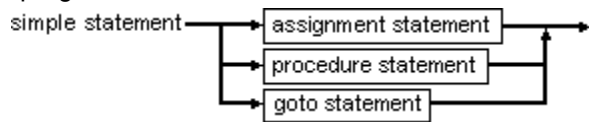


There are two basic types of statements:

- Simple statements
- Structured statements

Simple statements

Simple statements can either assign a value, activate a procedure or function, or transfer the running program to another statement in the code.



The simple statements that Object Pascal supports are:

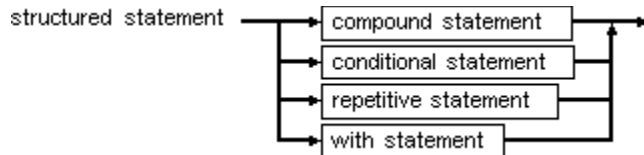
[Assignment \(:=\) statements](#)

[Goto statements](#)

[Procedure statements](#)

Structured statements

Structured statements are constructs composed of other statements that are to be executed sequentially, conditionally, or repeatedly.



The structured statements that Object Pascal supports are:

[Compound statements](#)

[Conditional statements](#)

[Loops](#)

[With statements](#)

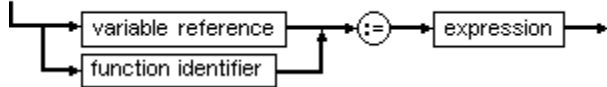
See also
Blocks

Assignment statements

See also [Example](#)

Assignment statements impart the value of the expression on the right side of the assignment operator to the identifier on the left side. The assignment operator, which separates the two sides of an assignment statement, is :=.

assignment statement



You can use assignment statements to do either of the following:

- Replace the current value of a variable with a new value specified by an expression
- Specify an expression whose value is returned by a function

Object type assignments

You can assign an instance of an object type an instance of any of its descendant types. Such an assignment constitutes a projection of the descendant onto the space spanned by its ancestor.

Note: Assigning an instance of an object does not initialize the instance.

See also

[Assignment compatibility](#)

[Assignment operator](#)

[Object types](#)

[Type compatibility](#)

Examples

`X := Y + Z;`

`Done := (I >= 1) and (I < 100);`

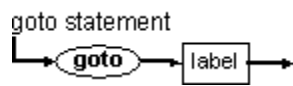
`Hue1 := [Blue, Succ(C)];`

`I := Sqr(J) - I * K;`

Goto statements

[See also](#)

A **goto** statement transfers program execution to the statement marked by the specified list.



When using **goto** statements, you must observe the following rules:

- The label referenced by the **goto** statement must be in the same block as the **goto** statement. You cannot jump into or out of a procedure or statement.
- Jumping into a structured statement from outside that structured statement can have undefined effects, although the compiler does not indicate an error.

Good programming practices recommend that you use **goto** statements as little as possible.

See also

[Goto \(reserved word\)](#)

[Scope](#)

Procedure statements

[See also](#)

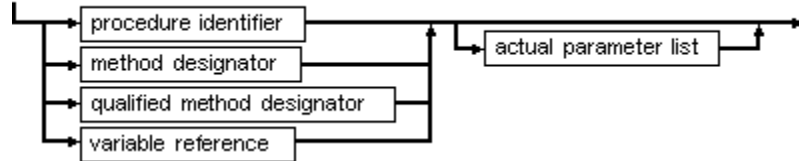
Procedure statements activate a procedure specified by one of the following:

- A procedure identifier
- A method designator
- A qualified-method designator
- A procedural-type variable reference

If the corresponding procedure declaration contains a list of formal parameters, then the procedure statement must have a matching list of actual parameters.

The actual parameters are passed to the formal parameters as part of the call.

procedure statement



See also

[Function calls](#)

[Method activation](#)

[Parameters](#)

[Procedural types](#)

[Procedure \(reserved word\)](#)

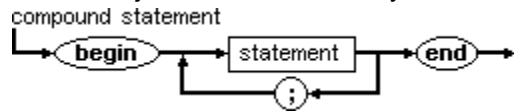
[Qualified-method activations](#)

[Variable reference](#)

Compound statements

[See also](#) [Example](#) [Language definition](#)

Compound statements specify that its constituent statements are to be executed in the same sequence as they are written. The compound statements are treated as one statement, which is crucial in context where Object Pascal allows only one statement.



The reserved words **begin** and **end** bracket the statements, and each statement is separated by a semicolon.

Example

The following code is an example of a compound statement:

```
begin
```

```
  Z := X;
```

```
  X := Y;
```

```
  Y := Z;
```

```
end;
```

See also

[Begin..end](#)

[Blocks](#)

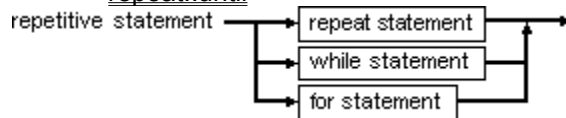
Loops

See also [Language definition](#)

Loops let you repeat one or more statements until or while a condition is met.

There are three kinds of loops:

- [for..to/downto..do](#)
- [while...do](#)
- [repeat..until](#)



The loop you want to use depends upon two criteria:

- The actions you want to perform
- How much you know about those actions prior to entering the loop

<u>Loop</u>	<u>When to use</u>
for	If you know exactly how many times you want the loop to repeat
while...do	If you want to test a condition before entering the loop
repeat...until	If you want the loop to execute at least once before the condition is tested

You can use the standard procedures [Break](#) and [Continue](#) to control the flow of a loop.

See also

[Conditional statements](#)

[Boolean expressions](#)

Conditional statements

[See also](#)

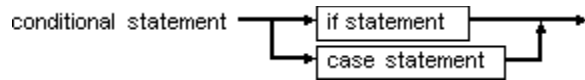
Conditional statements let you control whether certain expressions are evaluated. They test if a specific condition is met before executing the block of statements following the test.

There are two types of conditional statements:

- [if statements](#)
- [case statements](#)

Use **if** statements when you have only two possible choices.

Use **case** statements when you have many possible choices.



See also

[Boolean expressions](#)

[Loops](#)

Boolean expressions

[See also](#)

Boolean expressions evaluate to **True** or **False**. All loop and conditional statements depend upon Boolean expressions.

A Boolean expression compares two operands and produces a result that must be assigned to a variable of type Boolean.

The Boolean operators **and** and **or** work on pairs of Boolean values. Object Pascal supports two different models of code generation for these operators.

- Complete evaluation
- Short-circuit evaluation

The evaluation model is controlled through the `$B` compiler directive. In the default state `{B-}`, the compiler generates short-circuit evaluation code. In the `{B+}` state, the compiler generates complete evaluation.

See also

[Boolean operators](#)

[Boolean types](#)

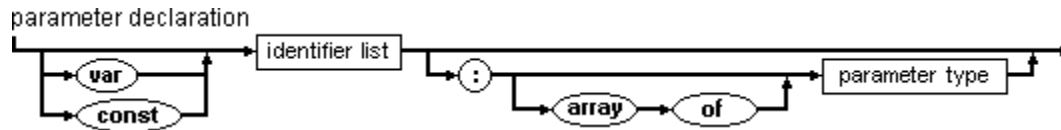
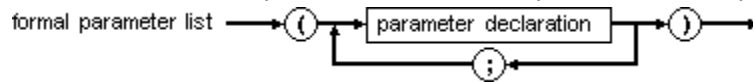
[Relational operators](#)

Parameters

[See also](#)

Parameters let you pass data to and receive data from a procedure or function.

The declaration of a procedure or function specifies a formal parameter list.



Each parameter declared in a formal parameter list is local to the procedure or function being declared. Parameters can be referred to by their identifier in the block statement associated with the procedure or function.

There are five types of parameters:

Value parameters Parameter group without a preceding **var** and followed by a type

Constant parameters Parameter group preceded by **const** and followed by a type

Variable parameters Parameter group preceded by **var** and followed by a type

Untyped parameters Parameter group preceded by **var** or **const** and not followed by a type

Open-array parameters Parameter group for array-type parameters

See also

[Function calls](#)

[Functions](#)

[Procedures](#)

Value parameters

[See also](#)

A formal value parameter is local to the declaring procedure or function, except it obtains its initial value from the corresponding actual parameter in the calling procedure or function.

Changes made to a formal value parameter do not affect the value of the actual parameter.

A value parameter's corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of file type or of any structured type that contains a file type.

The actual parameter must be assignment-compatible with the type of the formal value parameter.

See also

[Function calls](#)

[Parameters](#)

[Variable parameters](#)

Constant parameters

[See also](#)

A formal constant parameter is a local read-only variable that gets its value from the corresponding actual parameter.

Assignments to a formal constant parameter are not allowed, and likewise a formal constant parameter cannot be passed as an actual variable parameter to another procedure or function.

A constant parameter's corresponding actual parameter in a procedure statement or function must be an expression, and its value must not be of file type or of any structured type that contains a file type.

If you do not want a formal parameter to change its value during the execution of a procedure or function, use a constant parameter instead of a value parameter. Constant parameters protect against accidental assignments to a formal parameter.

For structured- and string-type parameters, the compiler generates more efficient code when constant parameters are used instead of value parameters.

See also

[Function calls](#)

[Parameters](#)

Variable parameters

[See also](#)

A variable parameter passes a variable to a procedure or function by reference. That is, the address of the parameter is passed so the value of the parameter can be accessed and modified.

In order for the actual parameter to be a variable parameter, the parameter must be passed by a variable reference. A variable reference is made by placing the var reserved word in the parameter list of the procedure or function declaration.

The formal variable parameter represents the actual variable during the activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected on the actual parameter.

Note File types can be passed only as variable parameters.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (you can bypass this restriction through untyped parameters).

If referencing an actual variable parameter involves indexing an array or finding the object of a pointer, these actions occur before the activation of the procedure or function.

See also

[Assignment compatibility](#)

[Function calls](#)

[Parameters](#)

[Untyped parameters](#)

Untyped parameters

[See also](#)

[Example](#)

When a formal parameter is an untyped parameter, the corresponding actual parameter can be any variable or constant reference, regardless of its type.

An untyped parameter declared using the var reserved word can be modified.

An untyped parameter declared using the const reserved word is read-only.

Within the procedure or function, the untyped parameter is typeless; that is, it is incompatible with variables of all other types, unless it is given a specific type through a variable typecast.

Untyped parameters give you greater flexibility, but they can be riskier to use because the compiler cannot verify valid operations.

Example

```
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;
```


Open-array parameters

[See also](#) [Example](#)

Open array parameters allow arrays of different sizes to be passed to the same procedure or function.

Declare formal parameters as open-array parameters using the following syntax:

array of T

T must be a type identifier, and the actual parameter must be a variable of type T, or an array variable whose element type is T.

Within the procedure or function, the formal parameter behaves as if it was declared as

array[0..N - 1] **of** T

where N is the number of elements in the actual parameter. The index range of the actual parameter is mapped onto the integers 0 to N - 1. If the actual parameter is a simple variable of type T, it is treated as an array with one element of type T.

You can access a formal open-array parameter only by element. Assignments to an entire open array are illegal.

You can pass open arrays to other procedures and functions only as an open-array parameter or as an untyped variable parameter.

Open-array parameters can be value parameters, constant parameters, or variable parameters, and their same restrictions hold true.

Note For an open-array value parameter, the compiler creates a local copy of the actual parameter within the procedure or function's stack frame. Therefore, be careful not to overflow the stack when passing large arrays as open-array value parameters. To ensure that the stack does not overflow, use **var** or **const** when passing open-array value parameters.

When passed as an open-character array, an empty string is converted to a string with one element containing a null character, so the statement `PrintStr("")` is identical to the statement `PrintStr(#0)`.

When the element type of an open-array parameter is Char, the actual parameter may be a string constant.

The following standard functions can be applied to open-array parameters:

Function	Return value
<u>Low</u>	Zero
<u>High</u>	The index of the last element in the actual array parameter
<u>SizeOf</u>	The size of the actual array parameter

Constructing open array parameters

You can construct an open-array parameter immediately, without declaring or assigning a variable or constant, by enclosing the desired array elements, separated by commas, between brackets. Therefore, instead of declaring and filling an array, you can construct and pass the array all at once:

```
MyProcedure([3, 2, 1900, 42]);
```

See also

[Array types](#)

[Parameters](#)

[Type-safe open arrays](#)

Example

```
procedure Clear(var A: array of Double); {assigns zero to each element of  
an array of Double}
```

```
var
```

```
  I: Integer;
```

```
begin
```

```
  for I := 0 to High(A) do A[I] := 0;
```

```
end;
```

```
function Sum(const A: array of Double): Double; {computes the sum of all  
elements in an array of Double}
```

```
var
```

```
  I: Integer;
```

```
  S: Real;
```

```
begin
```

```
  S := 0;
```

```
  for I := 0 to High(A) do S := S + A[I];
```

```
  Sum := S;
```

```
end;
```

```
procedure PrintStr(const S: array of Char); {allows string constants to be  
passed to the procedure }
```

```
var
```

```
  I: Integer;
```

```
begin
```

```
  for I := 0 to High(S) do
```

```
    if S[I] <> #0 then Write(S[I]) else Break;
```

```
end;
```

Type variant open-array parameters

[See also](#)

The new construct **array of const** allows an open array of objects of more than one type to be passed to a procedure or function in a type-safe manner. It makes it possible to declare a formatting routine which accepts any number of items of multiple types.

The following procedure declaration demonstrates the use of an **array of const** in the declaration of a string formatting function. The parameter `Args` accepts an open array containing any number of variables, each of any type:

```
procedure FmtStr(var Result: string; const Format: string; const Args: array  
  of const);
```

The compiler treats the construct **array of const** as identical to **array of TVarRec**.

See also

[Open array parameters](#)

[TVarRec type](#)

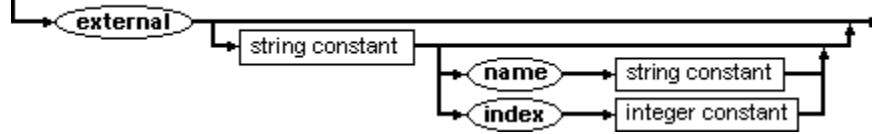
External declarations

See also [Example](#)

External declarations provide a means so you can:

- Interface with separately compiled procedures and functions written in assembly language
- Import procedures and functions from DLLs

external directive



External directives consisting only of the reserved word `external` are used in conjunction with `{$L filename}` directives to link with external procedures and functions implemented in `.OBJ` files.

External directives that specify a DLL name (and optionally an import name or import ordinal number) are used to import procedures and functions from DLLs.

The external directive takes the place of the declaration and statement parts in an imported procedure or function. Aside from this difference, imported procedures and functions behave like regular procedures and functions.

Examples

These are examples of external procedure declarations:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
```

```
procedure MoveLong(var Source, Dest; Count: Integer); external;
```

```
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

```
procedure FillLong(var Dest; Data: Longint; Count: Integer); external;
```

The following external declaration imports a function called `MessageBox` from the DLL called 'user32.dll' (part of the Windows API):

```
function MessageBox(HWnd: Integer; Text, Caption: PChar  
  Flags: Integer): Integer; stdcall;  
external 'user32.dll' name 'MessageBoxA';
```

See also

[Dynamic-Link Libraries](#)

[Functions](#)

[Procedures](#)

Method declarations

[See also](#) [Example](#)

The declaration of a method within an object type corresponds to a forward declaration of that method.

Somewhere after the object-type declaration and within the same scope as the object-type declaration, the method must be implemented by a defining declaration.

For procedure and function methods, the defining declaration takes the form of a normal procedure or function, but the procedure or function identifier is a qualified-method identifier.

For constructor methods and destructor methods, the defining declaration takes the form of a procedure method declaration, except that the **procedure** reserved word is replaced by a **constructor** or **destructor** reserved word.

Optionally, a method's defining declaration can repeat the formal parameter list of the method heading in the object type. The defining declaration's method heading must match exactly the order, types, and names of the parameters, and the type of the function result, if any.

Self Parameter

In the defining declaration of a method, there is always an implicit Self parameter, corresponding to a formal variable parameter that possesses the object type.

Within the method block, Self represents the instance whose method component was designated to activate the method. Therefore, any changes made to the values of the fields of Self are reflected in the instance.

Examples

{ Here are examples of method declarations }

```
procedure TRectangle.Intersect(var R: TRectangle);
```

```
begin
```

```
    if A.X < R.A.X then A.X := R.A.X;
```

```
    if A.Y < R.A.Y then A.Y := R.A.Y;
```

```
    if B.X > R.B.X then B.X := R.B.X;
```

```
    if B.Y > R.B.Y then B.Y := R.B.Y;
```

```
    if (A.X >= B.X) or (A.Y >= B.Y) then Init(0, 0, 0, 0);
```

```
end;
```

```
procedure TField.Display;
```

```
begin
```

```
    GotoXY(X, Y);
```

```
    Write(Name^, ' ', GetStr);
```

```
end;
```

```
function TNumField.PutStr(S: String): Boolean;
```

```
var
```

```
    E: Integer;
```

```
begin
```

```
    Val(S, Value, E);
```

```
    PutStr := (E = 0) and (Value >= Min) and (Value <= Max);
```

```
end;
```

See also

[Constructors and destructors](#)

[Methods](#)

[Object types](#)

Constructors and destructors

[See also](#) [Example](#)

Constructors and destructors are special methods that control construction and destruction of objects.

A class can have zero or more constructors and destructors for objects of the class type. Each is specified as a component of the class in the same way as a procedure or function method, except that the reserved words constructor and destructor begin each declaration instead of **procedure** and **function**. Like other methods, constructors and destructors can be inherited.

Constructors

Constructors are used to create and initialize new objects. Typically, the initialization is based on values passed as parameters to the constructor.

Contrary to an ordinary method, which must be invoked on an object reference, a constructor can be invoked on either a class reference or an object reference.

In order to create a new object, a constructor must be invoked on a class reference. When a constructor is invoked on a class reference, the following actions take place:

- Storage for a new object is allocated from the heap.
- The allocated storage is cleared. This causes the ordinal value of all ordinal type fields to become zero, the value of all pointer and class type fields to become **nil**, and the value of all string fields to become empty.
- The user-specified actions of the constructor are executed.
- A reference to the newly allocated and initialized object is returned from the constructor. The type of the returned value is the same as the class type specified in the constructor call.

When you invoke a constructor on an object reference, a new object is not allocated and cleared, and the constructor call does not return an object reference. Instead, the constructor operates on the specified object reference and executes only the user-specified actions given in the constructor's statement part. A constructor is typically invoked on an object reference only in conjunction with the **inherited** keyword to execute an inherited constructor.

The first action of a constructor is almost always to call an inherited constructor to initialize the inherited fields of the object. Following that, the constructor initializes the fields of the object that were introduced in the class. Since a constructor always clears the storage it allocates for a new object, all fields automatically have a default value of zero (ordinal types), **nil** (pointer and class types), empty (string types), or Unassigned (the Variant type). Unless a field's default value is nonzero, there is no need to initialize the field in a constructor.

If an exception occurs during execution of a constructor that was invoked on a class reference, the Destroy destructor is automatically called to destroy the unfinished object.

Like other methods, constructors can be virtual. When invoked through a class type identifier, as is usually the case, a virtual constructor is equivalent to a static constructor. However, when combined with object reference types, virtual constructors allow polymorphic construction of objects, that is, construction of objects whose types are not known at compile time.

Destructors

Destructors are used to destroy objects. When a destructor is invoked, the user-defined actions of the destructor are executed, and then the storage that was allocated for the object is disposed of. The user-defined actions of a destructor typically consist of destroying any embedded objects and releasing any resources that were allocated by the object.

The last action of a destructor is typically to call the inherited destructor to destroy the inherited fields of the object.

While it is possible to declare multiple destructors for a class, it is recommended that classes implement only overrides of the inherited Destroy destructor. Destroy is a parameterless virtual destructor declared in TObject, and since TObject is the ultimate ancestor of every class, the Destroy destructor is guaranteed to be available for any object.

If an exception occurs during execution of a constructor, the `Destroy` destructor is invoked to destroy the unfinished object. This means that destructors must be prepared to handle destruction of partially constructed objects. Since a constructor sets all fields of a new object to null values before executing any user-defined actions, any class-type or pointer-type fields in a partially constructed object are guaranteed to be **nil**. A destructor should therefore always check for **nil** values before performing operations on class-type or pointer-type fields.

Making a call to a `Free` method is a convenient way of checking for **nil** before invoking `Destroy` on an object reference. By calling `Free` instead of `Destroy` for any class-type fields, a destructor is automatically prepared to handle partially constructed objects resulting from constructor exceptions. For that same reason, direct calls to `Destroy` are not recommended.

Examples

The following example is the constructor and destructor for TShape.

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;
```

```
destructor TShape.Destroy;
```

```
begin
```

```
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
```

```
end;
```

```
procedure TObject.Free;
```

```
begin
```

```
  if Self <> nil then Destroy;
```

```
end;
```

See also

[Instantiating objects](#)

[Method declarations](#)

[Object types](#)

Indirect unit references

[See also](#) [Example](#)

The **uses** clause in a module need name only the units used directly by that module.

However often a module is directly dependent on another module. To compile a module, the compiler must be able to locate all units the module depends upon, directly or indirectly.

When you make changes to the interface part of a unit, you must recompile all other units that use the changed unit. If you use Project|Build All, the compiler does this for you.

If changes are made only to the implementation or the initialization part, however, you do not need to recompile other units that use the changed unit.

Note: For users of C and other languages: The **uses** clauses of a C++Builder program provides the "make" logic information traditionally found in make or project files of other languages. With the **uses** clause, C++Builder can build all the dependency information into the module itself and reduce the chance for error.

C++Builder can tell when the **interface** part of a unit has changed by computing a unit version number when the unit is compiled.

See also

[Compiling, building and running projects](#)

[Unit](#)

[Uses](#)

Example

{ The following is an example of units being dependent upon each other.
Notice that Unit2 directly depends on Unit1 and Prog is directly dependent upon Unit2. }

```
program Prog;  
uses Unit2;  
const a = b;  
begin  
end.
```

```
unit Unit2;  
interface  
uses Unit1;  
const b = c;  
implementation  
end.
```

```
unit Unit1;  
interface  
const c = 1;  
implementation  
const d = 2;  
end.
```

Circular unit references

[See also](#) [Example](#)

Circular unit references occur when you have mutually dependent units.

Mutually dependent units occur when you place a **uses** clause in the implementation part of a unit, essentially hiding the inner details of the unit referenced in the **uses** clause; the referenced unit is private and not available to the program or unit using the unit it is referenced in.

Because C++Builder can compile complete interface parts, two units can refer to each other in the **uses** clause in their implementation part. The compiler accepts a reference to a partially compiled unit in the implementation part of another unit, as long as neither unit's interface part depends upon the other. Therefore, the units follow Pascal's strict rules for declaration order.

If the interface parts are interdependent, C++Builder generates a circular unit-reference error.

Mutually dependent units can be useful in special situations, but use them judiciously. If you use them when they are not needed, they can make your program harder to maintain and more susceptible to errors.

See also

Unit

Uses

Example

{ The following program demonstrates how two units can "use" each other: }

```
program Circular;
{ Display text using WriteXY }

uses
  WinCrt, Display;

begin
  ClrScr;
  WriteXY(1, 1, 'Upper left corner of screen');
  WriteXY(1000, 1000, 'Way off the screen');
  WriteXY(81 - Length('Back to reality'), 15, 'Back to reality');
end.
```

```
unit Display;
{ Contains a simple video display routine }
```

```
interface
```

```
procedure WriteXY(X, Y: Integer; Message: String);
```

```
implementation
```

```
uses
  WinCrt, Error;
```

```
procedure WriteXY(X, Y: Integer; Message: String);
```

```
begin
  if (X in [1..80]) and (Y in [1..25]) then
    begin
      GoToXY(X, Y);
      Write(Message);
    end

    else
      ShowError('Invalid WriteXY coordinates');
  end;
end.
```

```
unit Error;
{ Contains a simple error-reporting routine }
```

```
interface
```

```
procedure ShowError(ErrMsg: String)
```

```
implementation
```

```
uses Display;
```

```
procedure ShowError(ErrMsg: String);
```

```
begin
  WriteXY(1, 25, 'Error: ' + ErrMsg);
end;
```

end.

Heap manager

[See also](#)

Windows supports dynamic memory allocations on two different heaps:

- The global heap
- The local heap

C++Builder includes a heap manager that implements the following standard procedures:

- New
- Dispose
- GetMem
- FreeMem

The heap manager uses the global heap for all allocations. Because the global heap has a system-wide limit of 8192 memory blocks (which is fewer than some applications might require), the heap manager includes a segment sub-allocator algorithm to enhance performance and allow a substantially larger number of blocks to be allocated.

So that the segment addresses of the blocks do not change, global blocks are locked using GlobalLock immediately after they are allocated, and not unlocked until immediately before they are deallocated.

In Windows standard and 386 enhanced modes, you can move fixed blocks in physical memory to make room for other memory allocation requests, so there is no performance penalty associated with using the C++Builder heap manager.

In Windows real mode, however, a fixed block must remain fixed in physical memory. This precludes the Windows memory manager from moving it so it can allocate other blocks.

If your application is to run in real mode, consider using the memory-management services provided by Windows when allocating dynamic memory blocks.

See also

DLLs

Exit procedures

[See also](#) [Example](#)

Exit procedures give you control over a program's termination process. This is useful when you want to carry out specific actions before a program terminates; a typical example is updating and closing files.

There are three types of application termination:

- Normal termination
- Termination through a call to Halt
- Termination because of a run-time error (C++Builder provides exception handling to handle run-time errors without terminating your application.)

To install an exit procedure, use the ExitProc pointer variable.

An exit procedure takes no parameters.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures making it possible for units, as well as programs, to install exit procedures. Some units install an exit procedure as part of their initialization code and then rely on that specific procedure to be called to clean up after the unit.

The procedures on the exit chain are executed in reverse order of installation. This ensures that the exit code of one unit is not executed before the exit code of any units that depend upon it.

To keep the exit chain intact, you must do the following:

- Save the current contents of ExitProc before changing it to the address of your own exit procedure
- Reinstall the saved value of ExitProc in the first statement in your exit procedure

The termination routine in the run-time library continues 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 will not be called again.

An exit procedure can learn the cause of termination by examining the ExitCode integer variable and the ErrorAddr pointer variable. ExitCode and ErrorAddr will have the following values depending on the type of termination.

Variable	Normal Termination	Halt	Run-Time Error
ExitCode	Zero	Value passed to Halt	Error code
ErrorAddr	nil nil	Address of the error statement	

The last exit procedure (the one installed by the run-time library) closes the Input and Output files. If ErrorAddr is not **nil**, it outputs a run-time error message.

Error messages

If you want to present run-time error messages yourself, install an exit procedure that examines ErrorAddr and outputs a message if it is not **nil**. In addition, before returning, make sure to set ErrorAddr to **nil**, so that the error is not reported again by other exit procedures.

Once the run-time library has called all exit procedures, it returns to Windows, passing the value stored in ExitCode as a return code.

See also

DLLs

Exception handling

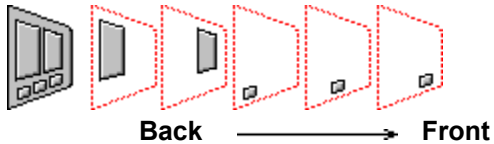
Example

{The following example demonstrates a skeleton method of implementing an exit procedure:}

```
program TestExit;  
var  
    ExitSave: Pointer;  
  
procedure MyExit;  
begin  
    ExitProc := ExitSave; { Always restore old vector first }  
end;  
begin  
    ExitSave := ExitProc;  
    ExitProc := @@MyExit;  
end.
```

Changing the Z-order of components

When a form contains overlapping components, the plane containing the last-added component always lies in front of the plane in which any previous components exist. In other words, overlapping objects on a form exist in layers.



Perhaps you may not add components in the order in which you want them to appear. You might want to move one component behind another. To do this, you must change the position of its visual layer. These visual layers on a form are known as the z-order, because these components lie on the z-axis (depth) of the layout. This determines what appears in front of (or on top of) what.

This is extremely useful when adding graphics or shapes that you want to appear in the background of your form.

To change the z-order of a component,

1. Select the component.
2. Choose either Edit|Bring to Front or Edit|Send to Back depending on which way you want to move the component.

Note: Windowed and non-windowed components have their own distinct z-order logic. You cannot include a non-windowed control, such as a Label or Shape component, in the z-order of a windowed control such as a button.

Direct memory access

Object Pascal implements three predefined arrays which are used to directly access memory.

- Mem
- MemW
- MemL

Each component of Mem is a byte, each component of MemW is a Word, and each component of MemL is a Longint.

The Mem arrays use a special syntax for indexes:

- Two expressions of the integer type Word, separated by a colon, are used to specify the segment base and offset of the memory location to access.

Here are two examples:

```
Mem[Seg0040:$0049] := 7;      {stores the value 7 in the byte at $0040:$0049}
Data := MemW[Seg(V):Ofs(V)]; {moves the Word value stored in the first 2
    bytes of the variable V into the variable Data}
```

Calling conventions

[See also](#)

Object Pascal provides four directives that allow you to specify the calling conventions to be used for passing parameters to procedures and functions. The default calling convention is always register.

Calling conventions differ in three areas:

- Order of passing parameters
- Responsibility for removing parameters from the stack ("cleanup")
- Use of registers for passing parameters

The **register** and **pascal** conventions pass parameters from left to right, that is the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl** and **stdcall** 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 must remove parameters from the stack when the call returns. The **register** convention uses up to three CPU registers to pass parameters, whereas the other conventions always pass all parameters on the stack.

The calling conventions are summarized in the following table.

Directive	Order	Cleanup	Registers
register	Left-to-right	Function	Yes
pascal	Left-to-right	Function	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Function	No

The **register** convention is by far the most efficient, since it often avoids the creation of a stack frame. The **pascal** and **cdecl** conventions are mostly useful for calling routines in dynamic-link libraries written in C, C++, or other languages. The **stdcall** convention is used for calling Windows API routines.

See also

Procedures

Functions

Precedence of operators

[See also](#)

Operators are symbols or reserved words used to indicate that some operation is to be performed on one or more pieces of data.

The precedence of the Object Pascal operators is:

Operators	Precedence	Category
@ not	First (high)	Unary operators
* / div mod as and shl shr	Second	Multiplicative and type casting operators
+ - or xor	Third	Additive operators
= <> < >	Fourth (low)	Relational, set membership, and type comparison operators

Rules of precedence

1. An operand between two operators of different precedence is bound to the operator with higher precedence.
2. An operand between two equal operators is bound to the one on its left.
3. Expressions within parentheses are evaluated before being treated as a single operand.

See also

[@ operator](#)

[Assignment operator](#)

[Binary arithmetic operators](#)

[Bitwise operators](#)

[Boolean operators](#)

[Character-pointer operators](#)

[Relational operators](#)

[Set operators](#)

[String operator](#)

[Unary arithmetic operators](#)

[Variant operators](#)

Binary arithmetic operators

See also [Operators](#)

Binary arithmetic operators perform arithmetic operations on two operands.

The binary arithmetic operators are:

Operator	Operation	Operand types	Result type
+	Addition	integer type	integer type
		real type real type	
-	Subtraction	integer type	integer type
		real type real type	
*	Multiplication	integer type	integer type
		real type real type	
/	Division	integer type	real type
		real type real type	
div	Integer division	integer type	integer type
mod	Modulus	integer type	integer type

Any operand whose type is a subrange of an ordinal type is treated as if it were of the ordinal type.

If both operands of a **+**, **-**, *****, **div**, or **mod** operator are of an integer type, the result type is of the common type of the two operands.

If both operands of a **+**, **-**, or ***** operator are of a real type, the result type is Extended.

If the operand of the sign identity or sign negation operator is of an integer type, the result is the same integer type. If the operand is of a real type, the type of the result is Extended.

The value of X/Y is always of type Extended regardless of the operand types. A run-time error occurs if Y is 0. You can handle that run-time error using exceptions.

The value of $I \text{ div } J$ is the mathematical quotient of I / J , rounded in the direction of 0 to an integer-type value. An error occurs if J is 0. You can handle that run-time error using exceptions.

The **mod** operator returns the remainder obtained by dividing its two operands:

$$I \text{ mod } J = I - (I \text{ div } J) * J$$

The sign of the result of **mod** is the same as the sign of I . An error occurs if J is 0.

Note: The **+**, **-**, and ***** operators can also be used as set operators, character-pointer operators, or unary operators. The **+** operator is also the string operator.

See also

Types

Is operator

[See also](#)

The **is** operator is used to perform dynamic type checking. Using the **is** operator, it is possible to check whether the actual (run-time) type of an object reference belongs to a particular class. The syntax of the **is** operator is

```
ObjectRef is ClassRef
```

where ObjectRef is an object reference and ClassRef is a class reference. The **is** operator returns a boolean value. The result is True if ObjectRef is an instance of the class denoted by ClassRef or an instance of a class derived from the class denoted by ClassRef. Otherwise, the result is False. If ObjectRef is **nil**, the result is always False. If the declared types of ObjectRef and ClassRef are known not to be related, that is if the declared type of ObjectRef is known not to be an ancestor of, equal to, or a descendant of ClassRef, the compiler will report a type mismatch error.

The **is** operator is often used in conjunction with an **if** statement to perform a guarded typecast. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

Here, if the **is** test is True, it is safe to typecast ActiveControl to be of class TEdit.

The rules of operator precedence group the **is** operator with the relational operators (=, <>, <, >, <=, >=, and in). This means that when combined with other boolean expressions using the **and** and **or** operators, **is** tests must be enclosed in parentheses:

```
if (Sender is TButton) and (TButton(Sender).Tag <> 0) then ...;
```

See also

[Conditional statements](#)

As operator

[See also](#)

The **as** operator is used to perform checked typecasts. The syntax of the **as** operator is

```
ObjectRef as ClassRef
```

where ObjectRef is an object reference and ClassRef is a class reference. The resulting value is a reference to the same object as ObjectRef, but with the type given by ClassRef. When evaluated at run time, ObjectRef must be **nil**, an instance of the class denoted by ClassRef, or an instance of a class derived from the class denoted by ClassRef. If neither of these conditions are True, an exception is raised. If the declared types of ObjectRef and ClassRef are known not to be related, that is if the declared type of ObjectRef is known not to be an ancestor of, equal to, or a descendant of ClassRef, the compiler will report a type mismatch error.

The **as** operator is often used in conjunction with a **with** statement, for example,

```
with Sender as TButton do  
  begin  
    Caption := '&Ok';  
    OnClick := OkClick;  
  end;
```

The rules of operator precedence group the **as** operator with the multiplying operators (*****, **/**, **div**, **mod**, **and**, **shl**, and **shr**). This means that when used in a variable reference, an **as** typecast must be enclosed in parentheses:

```
(Sender as TButton).Caption := '&Ok';
```

See also

[Variable typecasting](#)

[With statements](#)

Unary arithmetic operators

Operators

Unary arithmetic operators denote the sign of the operand.

The unary arithmetic operators are:

Operator	Operation	Operand types	Result type
+	Sign identity	integer type	integer type
		real type	real type
-	Sign negation	integer type	integer type
		real type	real type

Note: Any operand whose type is a subrange of an ordinal type is treated as if it were of the ordinal type.

If the operand of a + or - operator is of a real type, the result type is Extended.

Note: The + and - operators can also be used as set operators, character-pointer operators, or binary operators. The + operator is also the string operator.

Bitwise operators

Operators

The bitwise operators change the bit values for an integer.

The bitwise operators are:

Operator	Operation	Operand types	Result type
not	Bitwise negation	integer type	integer type
and	Bitwise and	integer type	integer type
or	Bitwise or	integer type	integer type
xor	Bitwise xor	integer type	integer type
shl	Bitwise shift left	integer type	integer type
shr	Bitwise shift right	integer type	integer type

not reverses bit values. For example, if the bit is 1, **not** changes it to a 0.

not is a unary operator. If the operand of the **not** operator is of an integer type, the result is of the same integer type.

The bitwise operators **and**, **or**, and **xor** perform Boolean operations between corresponding bits.

If both operands of an **and**, **or**, or **xor** operator are of an integer type, the result type is the common type of the two operands.

The operations $I \text{ shl } J$ and $I \text{ shr } J$ shift the value of I to the left or to the right by J bits. The result type is the same as the type of I .

not, **and**, **or**, and **xor** are also Boolean operators.

Boolean operators

See also [Operators](#)

Boolean operators use Boolean logic to evaluate expressions.

The Boolean operators are:

Operator	Operation	Operand types	Result type
not	negation	Boolean	Boolean
and	logical and	Boolean	Boolean
or	logical or	Boolean	Boolean
xor	logical xor	Boolean	Boolean

not takes a Boolean value and inverts it. For example, **not** True is False. **not** is a unary operator.

and Boolean expressions returns True if both operands evaluate to True.

or returns True if either or both operand is True.

xor returns True if one, but not both, of the operands is True. If both operands are True the expression evaluates to False.

The **and** and **or** operators work on pairs of Boolean values and Object Pascal supports two different models of code generation for these operators.

- Complete evaluation
- Short-circuit evaluation

The evaluation model is controlled through the `$B` compiler directive. In the default state `{B-}`, the compiler generates short-circuit evaluation code. In the `{B+}` state, the compiler generates complete evaluation.

not, **and**, **or**, and **xor** are also [bitwise operators](#).

See also

[Boolean expressions](#)

[Boolean types](#)

String operator

See also [Operators](#)

The + operator concatenates two strings.

Operator	Operation	Operand types	Result type
+	concatenation	string type, Char type, or packed string type	string type

The result is compatible with any string type (but not with Char and packed string types).

If both operands are short strings and the result is longer than 255 characters, the result is truncated after character 255. If either operand is a long string, the result is also a long string.

Note: The + operator can also be used as a set operator, unary operator, character-pointer operator, or binary operator.

See also

[Relational operators](#)

Character-pointer operators

Operators

The plus (+) and minus (-) operators can increment and decrement a character pointer value, and the minus (-) operator can calculate the distance (difference) between two character pointers.

The minus operator can calculate the distance (difference) between the offset parts of two character pointers.

Assuming that P and Q are values of type PChar and I is a value of type Integer, these constructs are allowed:

Construct	Result
$P + I$	Add I to P
$I + P$	Add I to P
$P - I$	Subtract I from P
$P - Q$	Subtract Q from P

The operations $P + I$ and $I + P$ add I to the address given by P, producing a pointer that points I characters after P.

The operation $P - I$ subtracts I from the address given by P, producing a pointer that points I characters before P.

The operation $P - Q$ computes the distance between Q (the lower address) and P (the higher address), resulting in a value of type Integer that gives the number of characters between Q and P.

This operation assumes that P and Q point within the same character array. If the two character pointers point into different character arrays, the result is undefined.

Set operators

Operators

Set operators are used to find the union, difference or intersection of two sets, or to test set membership.

The set operators are:

Operator	Operation	Operand types
+	Union	compatible set types
-	Difference	compatible set types
*	Intersection	compatible set types
in	Member of	left operand: any ordinal T; right operand: set whose base type is compatible with T.

The results of set operations conform to the rules of set logic:

- An ordinal value C is in $A + B$ only if C is in A or B.
- An ordinal value C is in $A - B$ only if C is in A and not in B.
- An ordinal value C is in $A * B$ only if C is in both A and B.

If the smallest ordinal value that is a member of the result of a set operation is A and the largest is B, the type of the result is `set of A..B`. The results of the **+**, **-**, and ***** operators are sets; the result of the **in** operator is a Boolean.

Relational operators

[See also](#) [Operators](#)

Relational operators compare operands and return a Boolean value based on the result.

Operator	Operation	Result type	Operand types
=	Equal	Boolean	compatible simple, class, class reference, pointer, set, string, packed string, or variant types
<>	Not equal	Boolean	compatible simple, class, class reference, pointer, set, string, packed string, or variant types
<	Less than	Boolean	compatible simple, string, packed string, PChar, or variant types
>	Greater than	Boolean	compatible simple, string, packed string, PChar, or variant types
<=	Less or equal	Boolean	compatible simple, string, packed string, PChar, or variant types
>=	Greater or equal	Boolean	compatible simple, string, or packed string, PChar, or variant types
<=	Subset of	Boolean	compatible set types
>=	Superset of	Boolean	compatible set types

See also

[Boolean expressions](#)

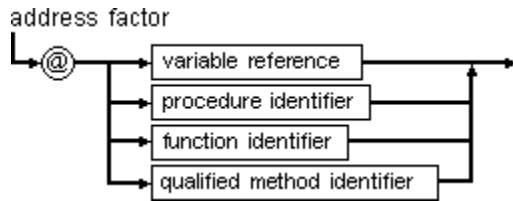
[Boolean operators](#)

[Type compatibility](#)

The @ ("at") operator: Pointer operation

See also [Example](#) [Operators](#)

The @ operator is used in an address factor to compute the address of a variable, procedure, function, or method.



The @ operator returns the address of its operand, that is, it constructs a pointer value that points to the operand.

You can create a pointer to a variable with the @ operator.

@ is a unary operator.

Special rules apply to the @ operator when used with a procedural variable.

The type of the value is the same as the type of `nil`, so it can be assigned to any pointer variable.

@ with	Returns a pointer to
variable	The variable.
value parameter	The stack location containing the actual value.
variable parameter	The actual parameter. The pointer type value is controlled by the \$T compiler directive.
procedure or function	The procedure's or function's entry point.
method	The method's entry point.

When the @ operator is applied to a variable reference, the type of the resulting pointer value is controlled through the `{$T}` compiler directive.

The type of the resulting pointer to a routine's entry point is always a `Pointer`.

The only use for a method or procedural pointer is to pass it to an assembly language routine.

The preferred way to reference a method is with a qualified method identifier.

Example

{The following example modifies a copy of the parameter.}

```
procedure ValueEx (X :Integer);
```

```
var
```

```
    ptr := ^integer;
```

```
begin
```

```
    ptr := @X;
```

```
    writeln(Ptr^);
```

```
    Ptr^ := 15;
```

```
end;
```

```
var
```

```
    Fred : integer;
```

```
begin
```

```
    Fred := 10;
```

```
    ValueEx (Fred);
```

```
    Writeln (Fred);    {10}
```

```
end.
```

{The following example modifies the actual parameter.}

```
procedure VarEx(var Y : integer);
```

```
var Ptr = ^integer;
```

```
begin
```

```
    Ptr := @Y;
```

```
    writeln (Ptr^);
```

```
    Ptr^ := 15;
```

```
end;
```

```
var Fred : integer;
```

```
begin
```

```
    Fred := 10;
```

```
    VarEx (Fred);
```

```
    writeln (Fred);    {15}
```

```
end.
```

See also

[Procedural values](#)

[Value parameters](#)

[Variable parameters](#)

Assignment operator

[See also](#)

[Example](#)

[Operators](#)

Syntax

Description

The assignment operator, `:=`, gives the value of expression (on the right side) to a variable of the same type (on the left side).

Example

X := Y;

Done := (I > 0) **and** (I < 100)

A[I] := A[I] + 1;

See also

[Assignment compatibility](#)

[Expression](#)

[Statement](#)

[Variable](#)

Variant operators

See also [Operators](#)

The `+`, `-`, `*`, `/`, **div**, **mod**, **shl**, **shr**, **and**, **or**, **xor**, and **not** operators support operands of type Variant. For binary operators, if one operand is of type Variant, the other operand is automatically converted to type Variant.

See also

[Variant types](#)

[Variant expressions](#)

Reserved words

[See also](#)

Reserved words have fixed meanings within the Object Pascal language: You cannot redefine them.

Object Pascal is not case sensitive, so you can use upper or lowercase letters in any combination for reserved words.

In the Borland manuals and in this Help system, reserved words appear in **boldface** type.

Here is an alphabetical listing of Object Pascal reserved words.

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

See also

[Standard directives](#)

Standard directives

[See also](#)

The Object Pascal standard directives have predefined meanings that can be redefined; however, this is not advised. Directives are used only in contexts where user-defined identifiers cannot occur.

In the Borland manuals and in this Help system, standard directives appear in **boldface** type.

Object Pascal is not case-sensitive, so you can use upper- or lowercase letters in any combination for directives.

Here is an alphabetical listing of the Object Pascal standard directives.

absolute

abstract

assembler

at

automated

cdecl

default

dispid

dynamic

external

forward

index

message

name

nodefault

override

pascal

private

protected

public

published

read

register

resident

stdcall

stored

virtual

write

The **private**, **protected**, **public**, **published**, and **automated** directives act as reserved words within class type declarations, but are otherwise treated as directives.

See also

[Reserved words](#)

Absolute

[Example](#)

[Standard directives](#)

Syntax

absolute address

Description

The standard directive **absolute** declares a variable that resides at a specific memory address.

You can do either of the following:

- Assign the variable directly to a specific address
- Declare the variable to reside at the same memory address as another variable

The first form directly specifies the address of the variable.

The second form declares a new variable on top of an existing variable (at the same address).

The variable declaration's identifier list can specify only one identifier when an **absolute** clause is present.

Example

var

Str: ShortString;

StrLen: Byte **absolute** Str;

Abstract

[Example](#)

[Standard directives](#)

Description

The **abstract** directive is used in an object definition to indicate that a virtual or dynamic method is not declared in the object in which it appears; its definition is then deferred to descendant classes. An abstract method in effect defines an interface, but not the underlying operation.

You cannot declare a method to be abstract unless it is first declared **virtual** or **dynamic**. An abstract method must not be called without being overridden. An override of an abstract method is identical to an override of a normal virtual or dynamic method, except that in the implementation of the overriding method, an **inherited** method is not available to call.

If you attempt to call an abstract method that has not been overridden, the run-time library procedure `Abstract` is called, and the program terminates with a run-time error.

Example

type

```
TMyObject = class  
  procedure Something; virtual; abstract;  
end;
```


Array

[See also](#)

[Example](#)

Syntax

array [index-type] **of** element-type

Description

The **array** reserved word defines an array type.

Several index types are allowed if they are separated by commas.

The element type can be any type, but the index type must be an ordinal type.

Example

type

```
IntList = array[1..100]    of Integer;  
CharData = array['A'..'Z'] of Byte;  
Matrix  = array[0..9, 0..9] of real;
```

See also

[Array type](#)

[Array-type constants](#)

[Indexes](#)

Asm

[See also](#)

[Reserved words](#)

Syntax

asm

```
AssemblerStmt <Separator AssemblerStmt>
```

end

where

- AssemblerStmt is an assembler statement.
- Separator is a semicolon, a new-line, or a Pascal comment.

Description

The **asm** reserved word accesses the built-in assembler.

When placing multiple assembler statements on a single line, separate them with semicolons. Assembler statements on separate lines do not require a semicolon.

In an **asm** statement, a semicolon does not indicate that the rest of the line is a comment. Comments must be Pascal style, using { and } or (* and *).

Register use

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 these registers:

EDI ESI

EBP EBX

An **asm** statement can freely modify these registers:

EAX EDX ECX

Except for EDI, ESI, EBP, and EBX registers, an **asm** statement can assume nothing about register contents.

See also

[Assembler directive](#)

[Comments](#)

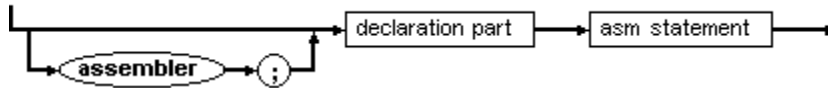
[External](#)

Assembler

See also [Standard directives](#)

Syntax

asm block



Description

The standard directive **assembler** lets you write complete procedures and functions in inline assembly language, without a **begin...end** statement.

Assembler causes the compiler to perform these code-generation optimizations:

- **Value Parameters:** The compiler does not generate code to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size is not 1, 2, or 4 bytes.
Within the procedure or function, such parameters must be treated as **var** parameters.
- **Function Result Variable:** The compiler does not allocate a function result variable, and a reference to the **@Result** symbol is an error.
String functions are an exception to the function-result optimization--they always have an **@Result** pointer allocated by the caller.
- **Stack Frame:** The compiler does not generate a stack frame for procedures and functions with no parameters or local variables.

The automatically generated entry and exit code for an assembler procedure or function looks like this:

```
PUSH    BP           ;Present if Locals <> 0 or Params <> 0
MOV     BP, SP      ;Present if Locals <> 0 or Params <> 0
SUB     SP, Locals  ;Present if Locals <> 0
.
.
.
MOV     SP, BP      ;Present if Locals <> 0
POP     BP          ;Present if Locals <> 0 or Params <> 0
RET     Params      ;Always present
```

If both **Locals** and **Params** are zero, there is no entry code, and the exit code is only of a **RET** instruction.

Functions using **assembler** must return their results as follows:

Function type	Return results
Ordinal	AL (8-bit values) AX (16-bit values) EAX (32-bit values)
Real	ST(0) on the coprocessor's register stack
Pointer	EAX
Short string	Temporary location pointed to by @Result

See also

[Asm](#)

Automated

See also [Standard directives](#)

The visibility rules for automated components are identical to those of public components. The only difference between automated and public components is that automation type information is generated for methods and properties that are declared in an **automated** section. This automation type information makes it possible to create OLE Automation Servers.

Note **automated** sections are typically only used in classes derived from the TAutoObject class defined in the OleAuto unit. For further details on creating OLE Automation Servers with C++Builder, see the C++Builder User's Guide.

The following restrictions apply to methods and properties declared in an **automated** section:

- For a method, the types of all method 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:

Byte, Currency, Double, Integer, Single, Smallint, **string**, TDateTime, Variant, WordBool.

Declaring methods or properties that use non-automatable types in an **automated** section results in an error.

- Method declarations must use the register calling convention (the default).
- Methods can be **virtual**, but not **dynamic**.
- Property declarations can only include access specifiers (**read** and **write**). No other specifiers (**index**, **stored**, **default**, **nodefault**) are allowed.
- Property access specifiers must list a method identifier. Field identifiers are not allowed.
- Property access methods must use the register calling convention.
- Property overrides (property declarations that don't include the property type) are not allowed.

A method or property declaration in an **automated** section can include an optional **dispid** directive, which must be followed by an integer constant that gives the OLE Automation dispatch ID of the method or property. If a **dispid** directive is not present, the compiler automatically picks a number one larger than the largest dispatch ID used by any method or property in the class and its ancestors. Specifying an already used dispatch ID in a **dispid** directive causes an error.

See also

Component visibility

private

protected

public

published

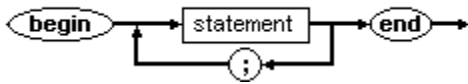
Begin ... End construct

[SeeAlso](#)

[Example](#)

[Reserved words](#)

Syntax



Description

The **begin** and **end** reserved words group a series of statements together into a compound statement.

The compound statement is then treated as a single statement.

Example

(* Compound statement used within an "if" statement *)

if First < Last **then**

begin

 Temp := First;

 First := Last;

 Last := Temp;

end;

See also
Statements

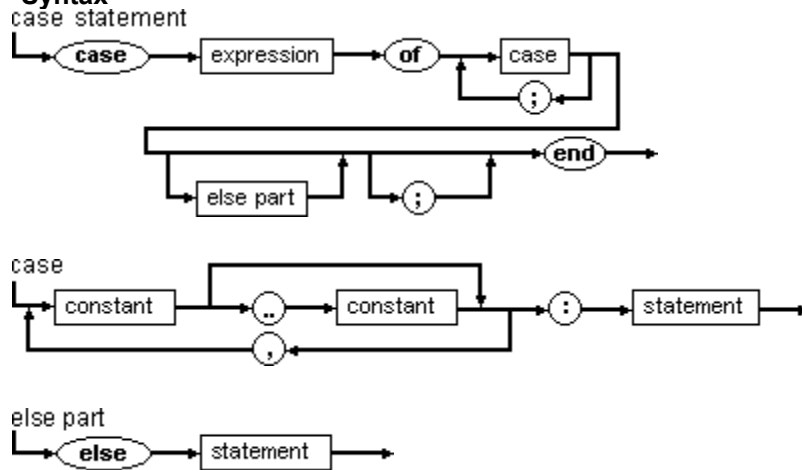
Case

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

Case statements are used to branch code depending on the results or values the code encounters.

A **case** statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called case constants) or with the reserved word **else**. The selector must be of an ordinal type, so string types are invalid selector types.

All case constants must be unique and of an ordinal type compatible with the selector type.

When the program enters a **case** statement, it evaluates each expression until a match is found. The program then performs the actions associated with that expression. If no match is found program defaults to the **else** statement. If there is no **else** part, execution continues with the next statement following the **case** statement.

Ranges in case statements must not overlap. So for example, the following case statement, is not allowed:

```
case MySelector of
  5: Writeln('Special case');
  1..10: Writeln('General case');
end;
```

Placing case constants in ascending order allows the compiler to optimize the case into jumps instead of calculating each time. For example, the compiler will turn this case statement into jumps:

```
case MySelector of
  1: Writeln('One');
  2: Writeln('Two');
  else Writeln('More');
end;
```

but this version will involve multiple calculations:

```
case MySelector of
  2: Writeln('Two');
  1: Writeln('One');
  else Writeln('More');
end;
```

Example

```
case Ch of
  'A'..'Z', 'a'..'z': WriteLn('Letter');
  '0'..'9':          WriteLn('Digit');
  '+', '-', '*', '/': WriteLn('Operator');
else
  WriteLn('Special character');
end;
```

See also

Else

Expression

Statement

Cdecl

See also

[Standard directives](#)

Syntax

procedure A; **cdecl**;

Description

The **cdecl** directive specifies that a procedure or function uses the C/C++ calling convention for passing parameters.

The C/C++ calling convention passes parameters from right to left, with parameters removed from the stack by the caller.

The C/C++ calling convention is mostly useful for calling routines exported from dynamic-link libraries (DLLs) written in C, C++, and other languages.

See also

[Calling conventions](#)

[pascal](#)

[register](#)

[stdcall](#)

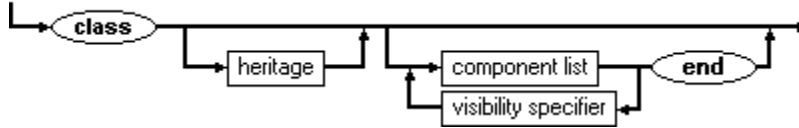
Class

See also

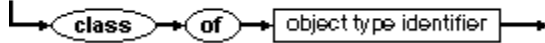
[Reserved words](#)

Syntax

object type



class reference type



Description

The **class** reserved word is used to declare an object type or class method. It is also used to define a class reference type.

An object type is a data structure that contains a fixed number of components. Each component is either a field (which contains data of a particular type); a method, which performs an operation on the object; or a property.

The declaration of a field specifies an identifier that names the field, and its data type.

The declaration of a method specifies a procedure, function, constructor, or destructor heading.

The definition of a property names the property and its access methods, and may provide information on how the property behaves during streaming.

An object type can inherit components from another object type. The inheriting object is a descendant and the object inherited from is an ancestor.

The domain of an object type consists of itself and all its descendants.

A class reference type is defined using the sequence of reserved words **class of** followed by the name of a class. A variable of a class reference type can be set at run time to refer either to the class named in the declaration or any of its subclasses.

See also

[Field and object component designators](#)

[Object-type scope](#)

[Object types](#)

Const

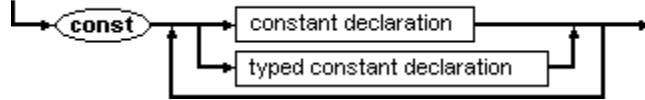
[See also](#)

[Example](#)

[Reserved words](#)

Syntax

constant declaration part



Description

The **const** reserved word defines an identifier whose value cannot change within the block containing the declaration. A constant identifier cannot be included in its own declaration.

C++Builder allows constant expressions.

Expressions used in constant declarations must be written such that the compiler can evaluate them at compile time.

See also

[Constant declarations](#)

[Expressions](#)

[Typed constants](#)

Examples

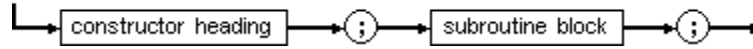
```
(* Constant Declarations *)  
const  
  MaxData = 1024 * 64 - 16;  
  NumChars = Ord('Z') - Ord('A') + 1;  
  Message = 'Hello world...';  
(* Typed constants *)  
const  
  identifier: type = value;  
  ...  
  identifier: type = value;
```

Constructor

See also [Reserved words](#)

Syntax

constructor declaration



constructor heading



Description

A constructor defines the actions associated with creating an object. It must be declared using the reserved word **constructor**. All C++Builder objects inherit at least a rudimentary constructor from TObject.

When invoked, the constructor returns a reference to a newly allocated and initialized instance of the class type.

See also

[Constructors and destructors](#)

[Destructor](#)

[Instantiating objects](#)

[Method](#)

[Object](#)

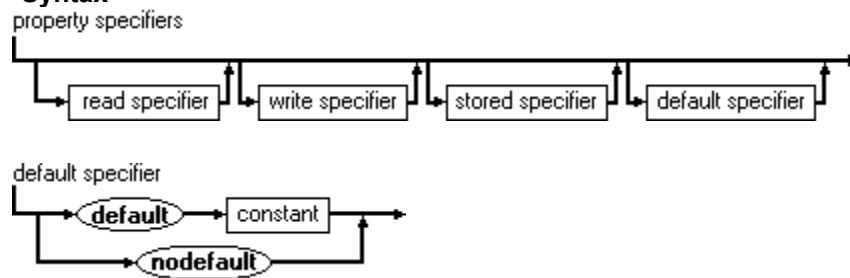
[Virtual](#)

Default

See also

[Standard directives](#)

Syntax



Description

The **default** directive is used to specify the default array property of an object.

When an array property is declared as the default you can access it using only the object name.

The **default** specifier is supported only for properties of ordinal types and small set types. If present in a property definition, **default** must be followed by a constant of the same type as the property.

If a property definition does not (or cannot) include a **default** or **nodefault** specifier, the results are the same as if a **nodefault** specifier had been included.

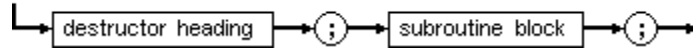
See also
[Nodefault](#)

Destructor

See also [Reserved words](#)

Syntax

destructor declaration



destructor heading



Description

A destructor defines the actions associated with destroying an object. It must be declared using the reserved word **destructor**.

When invoked a destructor will deallocate the memory that was allocated for the object by the constructor.

Destructors can be **virtual** and they seldom take any parameters.

See also

[Constructor](#)

[Constructors and destructors](#)

[Object](#)

[Virtual](#)

DispId

[See also](#)

[Standard directives](#)

Description

The **dispId** standard directive is used to specify an OLE automation dispatch ID for a method or property declared in an automated section of a class.

See also
[automated](#)

Do

[See also](#)

[Example](#)

[Reserved words](#)

The reserved word **do** is used in conjunction with **while**, **for**, **on**, and **with** statements to indicate which statements to execute while the condition holds true.

Example

```
while Ch = ' ' do Ch := GetChar;  
for Ch := 1 to 100 do Ch := GetChar;  
with Date[I] do month := 1;  
on <exception> do...
```


See also

Except

For

While

With

Dynamic

[See also](#)

[Standard directives](#)

Description

The **dynamic** directive makes a method dynamic. Dynamic methods are semantically identical to virtual methods. Virtual and dynamic methods differ only in the implementation of method call dispatching at run time; for all other purposes, the two types of methods can be considered equivalent.

In the implementation of virtual methods, the compiler favors speed of call dispatching over code size. The implementation of dynamic methods on the other hand favors code size over speed of call dispatching.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful only in situations where a base class declares a large number of virtual methods, and an application declares a large number of descendant classes with a small number of overrides of the inherited virtual methods.

See also
Methods

Else

[See also](#)

[Example](#)

[Reserved words](#)

The reserved word **else** is used as the default condition in **if**, **case**, and **try** statements.

Example

```
(* using if statement *)
if ParamCount <> 2 then
begin
  WriteLn('Bad command line');
  Halt(1);
end
else
begin
  ReadFile(ParamStr(1));
  WriteFile(ParamStr(2));
end;
(* using case statement *)
case Ch of
  'A'..'Z', 'a'..'z': WriteLn('Letter');
  '0'..'9':           WriteLn('Digit');
  '+', '-', '*', '/': WriteLn('Operator');
else
  WriteLn('Special character');
end;
```

See also

Case

If

Try

End

[See also](#)

[Example](#)

[Reserved words](#)

The reserved word **end** marks the end of a block. **End** can be used with

- **begin** to form compound statements
- **case** to form case statements
- **record** to declare record types
- **object** to declare object types
- **asm** to call the built-in assembler
- **except** to end an exception list
- **finally** to end a finally block

The final **end** of a module is followed by a period to denote that there is nothing after it.

See also

Asm

Begin

Case

Except

Finally

Object

Record

Examples

```
(* with begin to form compound statement *)
if First < Last then
begin
    Temp := First;
    First := Last;
    Last := Temp;
end;
(* with case statement *)
case Ch of
    'A'..'Z', 'a'..'z': WriteLn('Letter');
    '0'..'9':           WriteLn('Digit');
    '+', '-', '*', '/': WriteLn('Operator');
else
    WriteLn('Special character');
end;
(* in record type definitions *)
type
    MyClass = (Num, Dat, Str);
    Date = record
        D, M, Y: Integer;
    end;
    Facts = record
        Name: string[10];
        case Kind: MyClass of
            Num: (N: real);
            Dat: (D: Date);
            Str: (S: string);
        end;
(* in object type definitions *)
type
    Location = object
        X, Y: Integer;
        procedure Init(PX, PY: Integer);
        function GetX: Integer;
        function GetY: Integer;
    end;
(* with asm *)
asm
    mov ax,1
    mov cx, 100
end;
```

Except

[See also](#)

[Reserved words](#)

Syntax

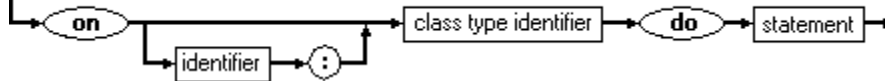
try statement



exception block



exception handler



Description

The **except** reserved word marks the beginning of the list of exception handlers in an exception-handling block.

The **except** part is a list of specific exceptions and responses to them, with each being an **on..do** statement. If none of the **on..do** statements apply to the current exception, the default exception handler in the **else** part is executed.

When an exception occurs in an exception-handling block, execution jumps immediately to the **except** part, where the application looks to each **on..do** statement until it finds one that applies to the specific exception raised. If no specific handler exists in the block, the application executes the default handler (specified with the **else** reserved word) if any.

Once a handler (either a specific one or the default handler) deals with the exception, the exception is considered handled, the exception object destroyed, and execution continues after the exception-handling block.

If no exception handler applies to the specific exception raised, execution leaves the block with the exception still unhandled.

See also

Else

Exception handling

Finally

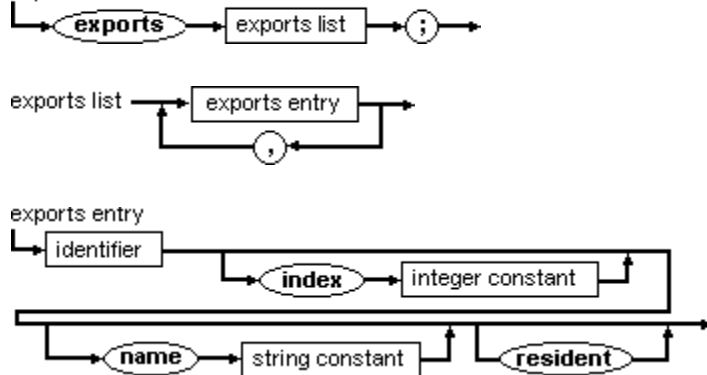
Try

Exports

[See also](#)

[Reserved words](#)

The **exports** reserved word is used in [DLLs](#) to list procedures and functions exported by that DLL.



You can use an **exports** clause anywhere and any number of times in a program's or library's declaration.

Each entry in an **exports** clause specifies the identifier of a procedure or function to be exported. The procedure or function to be exported must be declared before the **exports** clause appears.

You can precede the identifier in the exports clause with a unit identifier and a period; this is known as a fully qualified identifier.

An **exports** clause can also include

- An [index](#) clause
- A [name](#) clause

The quickest way to look up a DLL entry is by index.

A program can contain an **exports** clause, but it seldom does because Windows does not allow application modules to export functions for use by other applications.

See also

[Dynamic-linked libraries](#)

[Using DLLs](#)

[Writing DLLs](#)

External

[See also](#)

[Examples](#)

[Standard directives](#)

The **external** directive lets your program interface with separately compiled procedures and functions written in assembly language, or located in DLLs.

The **external** directive takes the place of the declaration and statement parts that would otherwise be present.

The external code for assembly language routines is linked with the Pascal unit or program through [\\$L filename](#) compiler directives.

For a complete discussion of importing external routines from DLLs, see [Accessing routines stored in DLLs](#).

Examples

{ The following lines import routines from an external assembly language file }

function GetMode: Word; **external**;

procedure SetMode(Mode: Word); **external**; {\$L CURSOR.OBJ}

{ The following line imports a function from a DLL. }

function GlobalAlloc(Flags: Word; Bytes: Longint): THandle; **external**
'KERNEL.DLL' **index** 15;

See also

DLLs

Functions

Procedures

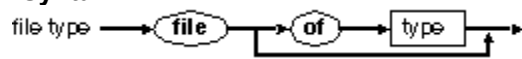
File

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

A **file** type consists of a linear sequence of data. Use the reserved word **of** to assign a **file** to a specific type. Files can be of any type except for type file or object.

If **of** and the component type are omitted, it is an untyped file.

- The predefined **file** type Text signifies a file containing printable ASCII characters organized into lines.

Example

```
(* File type declarations *)
type
  Person = record
    FirstName: string[15];
    LastName  : string[25];
    Address   : string[35];
  end;
  PersonFile = file of Person;
  NumberFile = file of Integer;
  SwapFile = file;
```

See also

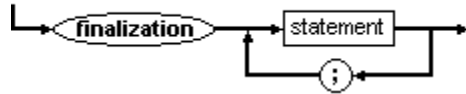
Of

Finalization

See also [Reserved words](#)

Syntax

finalization part



Description

The finalization part is optional and can only appear if a unit also has an [initialization part](#). The finalization part consists of the reserved word **finalization**, followed by a list of statements which finalize the unit. Finalization is the counterpart of initialization, and any resources (memory, files, etc.) acquired by a unit in its initialization part are typically released in the finalization part.

Unit finalization parts execute in the opposite order of initializations. 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 part is guaranteed to execute when the application shuts down. The finalization part must therefore be able to handle incompletely-initialized data, since, if an exception is raised, the initialization code might not execute completely.

See also

[Interface](#)

[Implementation](#)

[Initialization](#)

[Units](#)

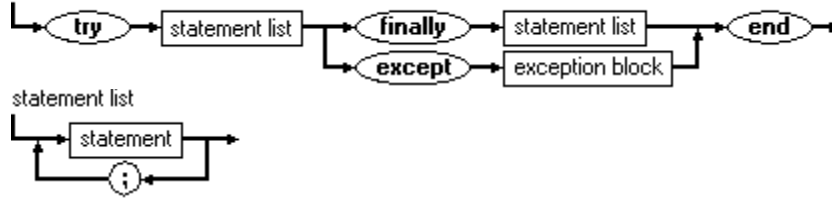
Finally

[See also](#)

[Reserved words](#)

Syntax

try statement



Description

The **finally** reserved word marks the section of a protected block that always executes, even if an exception occurs.

When an exception occurs, you might need to execute some cleanup code, such as releasing allocated resources, before handling the exception. The **try..finally** block enables you to do that.

All statements in a **try..finally** block execute normally unless an exception occurs, at which point execution jumps immediately to the statements in the **finally** part.

Note that a **try..finally** block does not handle particular exceptions. It just enables you to ensure that certain code in a block always executes, regardless of exceptions.

See also

Except

Exception handling

Try

Protecting resource allocations

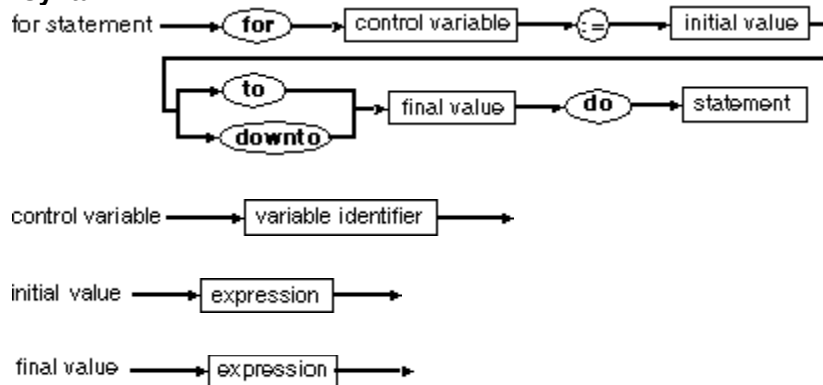
For ... To, For ... Downto

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

The **for** statement causes the statements after **do** to be executed once for each value between the initial value of the range and final value, inclusive. **For** loops are useful if you know beforehand exactly how many times you want the loop to be executed.

The control variable always starts off at initial value.

to Increments the control variable by 1 for each loop. The initial value must be less than the final value.

downto Decrements the control variable by 1 for each loop. The initial value must be greater than the final value.

The following rules apply to the control variable:

- It must be a variable identifier that is local in scope to the block containing the **for** statement.
- It must be of an ordinal type.

The initial and final values must be of a type assignment compatible with the ordinal type of the control variable.

After a **for** statement is executed, the value of the control variable is undefined, unless execution of the **for** statement was interrupted by a **goto** statement.

Example

```
(* for ... to, for ... downto *)
for I := 1 to ParamCount do
  WriteLn(ParamStr(I));
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];
      Mat[I, J] := X;
    end;
```

See also

[Goto statements](#)

[Loops](#)

[Ordinal types](#)

[Scope](#)

Forward

[See also](#)

[Example](#)

[Standard directives](#)

The standard directive **forward** allows you to declare a procedure or function without actually defining it.

From the point of the **forward** declaration, other procedures and functions can call the forwarded routine, making mutual recursion possible.

Somewhere after a **forward** declaration, you must define the procedure or function with a declaration that specifies the statement part of the routine.

The defining declaration can omit the parameter list from the procedure or function header.

A procedure's or function's defining declaration can be an **external** or **assembler** declaration; however, it cannot be another **forward** declaration.

Example

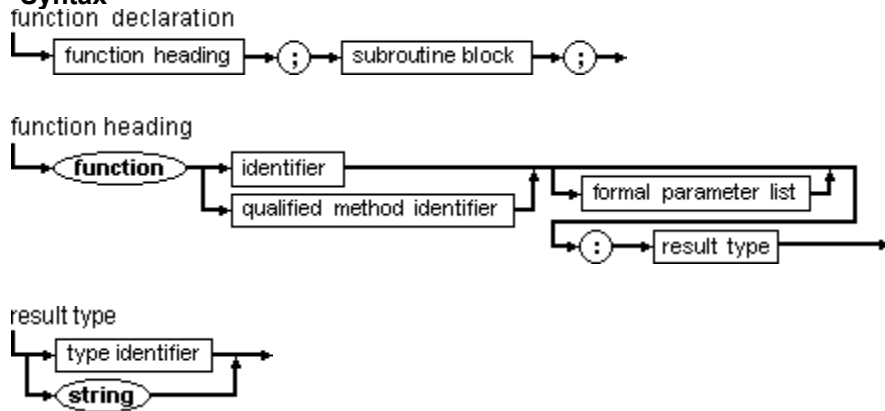
```
(* Forwarded procedure *)  
procedure Flip(N: Integer); forward;  
procedure Flop(N: Integer);  
begin  
  WriteLn('Flop');  
  if N > 0 then Flip(N - 1);  
end;  
procedure Flip;  
begin  
  WriteLn('Flip');  
  if N > 0 then Flop(N - 1);  
end;
```

See also
Assembler
External

Function

[See also](#) [Example](#) [Reserved words](#)

Syntax



Description

The reserved word **function** defines a block that computes and returns a value.

The **function** heading specifies the identifier for the function, the formal parameters (if any), and the function result type.

Functions can return values of any type except file types.

A function can have declaration parts following the function heading.

The **function** heading is followed by:

- Declarations of local variables, types, labels, constants, procedures, or functions
- Statements that execute when the function is called

The statement part must contain at least one statement that assigns a value to the function identifier; the result of the function is the last value assigned.

Instead of the declaration and statement parts, a **function** declaration can specify any of the following:

- forward declaration
- external declaration

Result variable in functions

Every function implicitly has a local variable `Result` of the same type as the function's return value.

Assigning to `Result` has the same effect as assigning to the name of the function. A function is activated by the evaluation of a function call. The function call gives the function's identifier and actual parameters, if any, required by the function.

In addition, however, you can refer to `Result` on the right side of an assignment statement, which refers to the current return value rather than generating a recursive function call.

See also

[Expression](#)

[Function calls](#)

[Ordinal](#)

[Parameters](#)

[Pointer](#)

[Real](#)

[String](#)

Example

```
(* Function declaration *)  
function UpCaseStr(S: string): string;  
var  
  I: Integer;  
begin  
  for I := 1 to Length(S) do  
    if (S[I] >= 'a') and (S[I] <= 'z') then  
      Dec(S[I], 32);  
      UpCaseStr := S;  
end;
```


Goto

[See also](#)

[Example](#)

[Reserved words](#)

Syntax

Description

The reserved word **goto** transfers program execution to the statement prefixed by the label referenced in the statement.

The label must be in the same block as the **goto** statement; it is not possible to jump out of a procedure or a function.

See also

[Label](#)

Example

```
label 1, 2;
```

```
goto 1
```

```
.
```

```
.
```

```
.
```

```
1: WriteLn ('Abnormal program termination');
```

```
2: WriteLn ('Normal program termination');
```

If ... Then ... Else

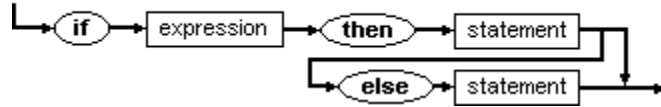
[See also](#)

[Example](#)

[Reserved words](#)

Syntax

if statement



Description

If, **then**, and **else** specify the conditions under which a statement will be executed.

If the Boolean expression after **if** is True, the statement after **then** is executed.

Otherwise, if the expression evaluates to False and the **else** part is present, the statement after **else** is executed. If the **else** part is not present, execution continues with the next statement following the **if** statement.

Note: No semicolon is allowed preceding an **else** clause.

See also

[Boolean types](#)

[Conditional statements](#)

[Else](#)

Example

```
(* if statements *)  
if (I < Min) or (I > Max) then I := 0;  
if x < 1.5 then  
  z := x + y  
else  
  z := 1.5;
```

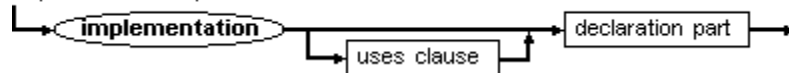
Implementation

See also [Reserved words](#)

The implementation part of a unit defines the block of all public procedures and functions declared in the [interface](#) part of the unit. It also declares constants, types, variables, procedures, and functions that are private.

Syntax

implementation part



Description

Declarations made in the implementation part of a unit are private and can be used only within this part of the unit. All constants, types, variables, procedures, and functions declared in the interface part are visible in the implementation part.

Implementations of procedures and functions declared in the interface part can be defined and referenced in any sequence in the implementation part.

A [uses](#) clause can appear in the implementation, immediately following the reserved word **implementation**. If you put a **uses** clause in the interface part of a unit, those units which are listed in the **uses** clause are not seen by the defining unit.

If any procedures are declared **external**, one or more \$L filename directive(s) should appear in the source file before the final [end](#) of the unit.

The [procedure/function](#) header in the implementation should be either of the following:

- Identical to the declaration in the interface
- In the [short form](#)

See also

[\\$L_filename](#)

[Interface](#)

[Initialization](#)

[Finalization](#)

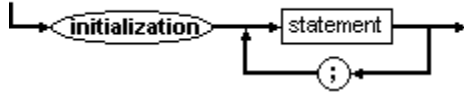
[Units](#)

Initialization

See also [Reserved words](#)

Syntax

initialization part



Description

The optional initialization part of a unit consists of the reserved word **initialization**, followed by a list of statements that initialize the unit.

The initialization parts of units used by a program are executed in the same order that the units appear in the uses clause of the main program.

See also
Finalization
Units

Short-form headers

[See also](#)

Short-form headers are procedures and functions declared in the **implementation** part that do not list their parameters. The parameters of a short-form header are previously listed in the **interface** part or are declared **forward** or as part of an object type.

For short-form headers, type the reserved word (**procedure** or **function**), followed by the routine identifier.

Routines local to the implementation part (not declared in the interface part) must have a complete procedure/function header.

See also
[Implementation](#)

Index

[See also](#)

[Example](#)

[Standard directives](#)

An **index** clause specifies an ordinal number for exporting procedures or functions from a dynamic-link library (DLL). If no **index** clause is used in an exports clause, the compiler assigns an ordinal number.

An **index** clause is included in an **exports** clause and consists of the word **index** followed by an integer constant between 1 and 32767.

Example

```
procedure ImportByOrdinal; external 'TESTLIB' index 5;
```

See also

[Dynamic-linked libraries](#)

[Forward](#)

[Using DLLs](#)

[Writing DLLs](#)

Inherited

[See also](#)

[Reserved words](#)

The reserved word **inherited** denotes the ancestor of the enclosing method's object type.

Inherited cannot be used within methods of an object type with no ancestor because it has no declaring object to inherit from.

See also
[Object types](#)

Inline

[Reserved words](#)

Description

The reserved word **inline** is not used in this version of Object Pascal. It remains reserved, however, for future use.

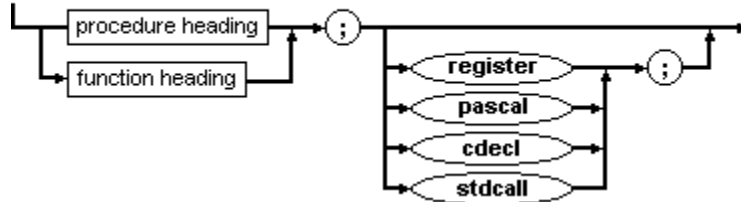
Interface

See also [Reserved words](#)

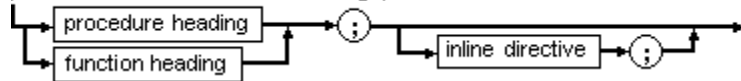
The interface part of a unit determines what is visible and accessible to any program (or other unit) using that unit.

Syntax

procedure and function
heading part



procedure and function heading part



Description

The interface part starts at the reserved word **interface**, which appears after the unit header, and ends before the reserved word **implementation**.

The interface part declares constants, types, variables, procedures, and functions that are public. That is, other programs or units can use them.

The interface part only lists the heading of a declared procedure or function. The block of the procedure or function follows in the [implementation](#) part. In effect, the procedure and function declarations in the interface part are like [forward declarations](#), although the **forward** directive is not used.

A [uses](#) clause can appear in the interface part. (If present, **uses** must immediately follow the reserved word **interface**).

See also

Implementation

Units

Label

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

The reserved word **label** declares placeholders that mark statements in the corresponding statement part. Control is transferred to a label via a **goto** statement.

Each label must mark only one statement.

In addition to an identifier, a digit sequence between 0 and 9999 can also be used as a label.

See also

[Goto](#)

Library

[See also](#)

[Reserved words](#)

· Syntax

·

Description

A dynamic-link library (DLL) starts with a **library** header.

The library header tells the compiler to produce an executable file with the extension .DLL instead of .EXE.

See also

[Exports clause](#)

[Import units](#)

[Index clause](#)

[Name clause](#)

[Writing DLLs](#)

MaxInt and MaxLongInt

MaxInt and MaxLongInt are predefined constants. In this version of Object Pascal, they have the same value.

- MaxInt contains the largest possible Integer (2,147,483,647).
- MaxLongInt contains the largest possible Longint (2,147,483,647).

Name

[See also](#) [Standard directives](#)

A **name** clause can be included in an exports clause. A **name** clause consists of the word **name** followed by a string constant.

When you use a **name** clause, the procedure or function is exported using the name specified by the string constant.

If no **name** clause is used, the procedure or function is exported by its identifier and is converted to all uppercase.

See also

[Using DLLs](#)

[Exports](#)

[Index](#)

Nil

[Reserved words](#)

Description

The reserved word **nil** denotes a [pointer](#) type constant that does not point to anything.

nil is compatible with all pointer types.

Nodefault

[See also](#)

[Standard directives](#)

Syntax

▪

Description

The **nodefault** directive controls the value that is considered a property's default value.

In a property declaration, **nodefault** is an optional specifier. If it is not included, the results are the same as if a **nodefault** specifier had been included.

See also
[Default](#)

Object

[See also](#)

Syntax

Description

The reserved word **object** is used to declare object types that conform to the object model used in previous versions of Borland and Turbo Pascal. New programs should use the reserved word **class** and the new object model. Object types declared using the reserved word **object** and the old object model may not have class methods, nor may they have properties as components.

An object type is a data structure that contains a fixed number of components.

Each component is either a field (which contains data of a particular type) or a method, which performs an operation on the object.

The declaration of a field specifies an identifier that names the field, and its data type.

The declaration of a method specifies a procedure, function, constructor, or destructor heading.

An object type can inherit components from another object type. The inheriting object is a descendant and the object inherited from is an ancestor.

The domain of an object type consists of itself and all its descendants.

See also

[Field and object component designators](#)

[Object-type scope](#)

[Object types](#)

Of

[See also](#)

[Example](#)

[Reserved words](#)

The reserved word **of** precedes a type in array, set, class, file type declarations, and in case statements.

See also

[Array](#)

[Case](#)

[File](#)

[Set](#)

Examples

```
(* array declaration *)
type
  IntList = array[1..100]    of Integer;
  CharData = array['A'..'Z'] of Byte;
  Matrix   = array[0..9, 0..9] of real;
(* Set types *)
type
  Day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  CharSet = set of Char;
  Digits = set of 0..9;
  Days = set of Day;
(* File type declarations *)
type
  Person = record
    FirstName: string[15];
    LastName  : string[25];
    Address   : string[35];
  end;
  PersonFile = file of Person;
  NumberFile = file of Integer;
  SwapFile = file;
(* case statement *)
case Ch of
  'A'..'Z', 'a'..'z': WriteLn('Letter');
  '0'..'9':          WriteLn('Digit');
  '+', '-', '*', '/': WriteLn('Operator');
else
  WriteLn('Special character');
end;
```

On

[See also](#)

[Reserved words](#)

Syntax

Description

The reserved word **on** defines responses to exceptions. **On** is always coupled with the reserved word **do** to form an entire exception handler.

The **except** part of a **try..except** block consists of a list of one or more **on..do** statements for the handling of specific exceptions.

See also

[Exception-handling statements](#)

[Responding to exceptions](#)

[Do](#)

[Except](#)

[Try](#)

Override

[Example](#)

[Standard directives](#)

The **override** directive is used to redefine a virtual or dynamic method.

When the **override** directive is included in the declaration of a method, the method overrides the inherited implementation of the method. An override of a virtual method must exactly match the order and types of the parameters, and the function result type (if any), of the original method.

Because virtual methods have two kinds of dispatching, VMT-based and dynamic, methods that override virtual and dynamic methods use the override directive instead of repeating **virtual** or **dynamic**.

Example

The following example uses **override** to replace the inherited procedure P.

```
type
  TAnObject = class
    procedure P; virtual;
  end;
  TAnotherObject = class(TAnObject)
    procedure P; override;
  end;
```

Packed

[See also](#)

[Reserved words](#)

Description

The reserved word **packed** in a structured type declaration tells the compiler to compress data storage, even at the cost of slower access to a component of a variable of this type.

However, it has no effect in C++Builder, since packing occurs automatically.

See also

[Structured types](#)

Pascal

[See also](#)

[Standard directives](#)

Description

The **pascal** directive specifies that a procedure or function uses the Pascal calling convention for passing parameters.

The Pascal calling convention passes parameters from left to right, with parameters removed from the stack by the function.

The Pascal calling convention is mostly useful for calling routines exported from dynamic-link libraries (DLLs) written in C, C++, and other languages.

See also

[Calling conventions](#)

[cdecl](#)

[register](#)

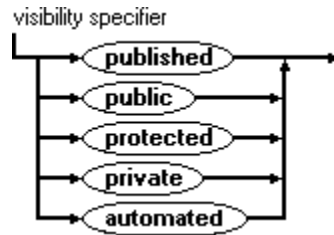
[stdcall](#)

Private

[See also](#)

[Standard directives](#)

Syntax



Description

The **private** directive is used within an object to denote a component declaration part.

- Inside the module, **private** component identifiers act like **public** component identifiers.
- Outside the module, **private** component identifiers are unknown and inaccessible.

Place related object types in the same module (or unit) so they can access each other's **private** components without making those **private** components known to other modules.

The scope of component identifiers declared as **private** are restricted to the module containing the object type declaration.

See also

[Objects](#)

[Protected](#)

[Public](#)

[Rules of scope](#)

[Object-type scope](#)

[Units](#)

Procedure

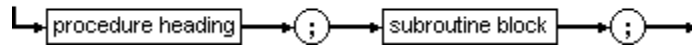
[See also](#)

[Example](#)

[Reserved words](#)

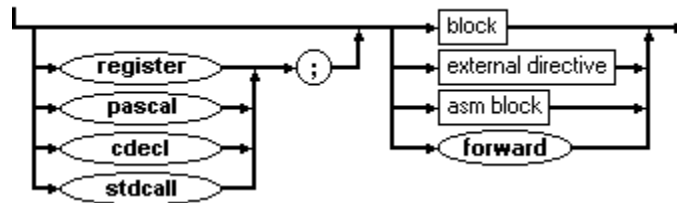
Syntax

procedure declaration



.

subroutine block



Description

Procedures let you nest additional blocks in the main program block. Each **procedure** declaration has a heading followed by a block of statements.

The **procedure** heading specifies the identifier for the procedure and the formal parameters (if any).

A **procedure** is activated by a procedure statement, which states the procedure's identifiers and actual parameters, if any.

The **procedure** heading is followed by:

- A declaration part that declares local objects
- The statements between **begin** and **end**, which specify what is to be executed when the procedure is called.

Note: If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure calls itself while being executed. This results in an endless loop.

Instead of the declaration and statement parts, a procedure declaration can specify any of the following directives:

- assembler
- external
- forward

Example

```
{ Procedure Declaration }  
procedure NumString(N: Integer; var S: string);  
var  
    V: Integer;  
begin  
    V := Abs(N);  
    S := '';  
    repeat  
        S := Chr(N mod 10 + Ord('0')) + S;  
        N := N div 10;  
    until N = 0;  
    if N < 0 then  
        S := '-' + S;  
end;
```

See also

[Functions](#)

[Parameters](#)

[Procedure statements](#)

[Procedural-type constants](#)

[Calling conventions](#)

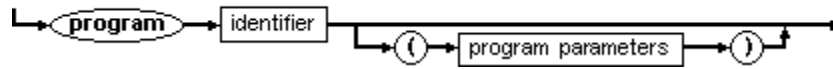
Program

[See also](#)

[Reserved words](#)

Syntax

program heading



Description

The reserved word **program** is placed at the top of a program and specifies the program's name.

See also

[Uses clause](#)

[Labels](#)

[Constants](#)

[Types](#)

[Variables](#)

[Procedures](#)

[Functions](#)

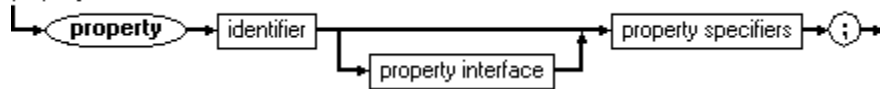
[Statements](#)

Property

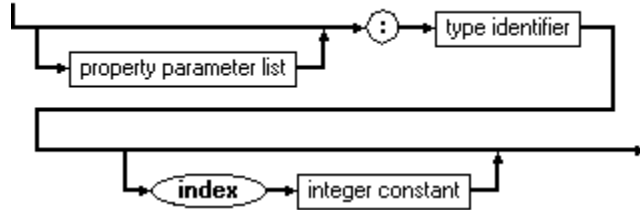
[See also](#)

Syntax

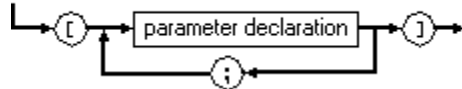
property definition



property interface



property parameter list



Description

The reserved word **property** enables you to declare properties. A property definition in a class declares a named attribute for objects of the class and the actions associated with reading and writing the attribute.

See also

Read

Write

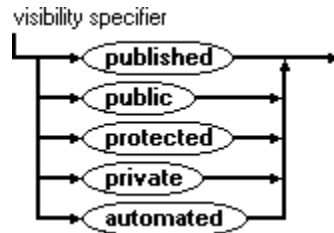
Stored

Object types

Protected

See also [Standard directives](#)

Syntax



Description

The **protected** directive is used in object type declarations.

Components declared as **protected** are accessible only to descendants of the declaring type.

Declaring a component as **protected** combines the advantages of **public** and **private** components.

As with **private** components, you can hide implementation details from end users. However, unlike **private** components, **protected** components are still available to programmers who want to derive new objects from your objects without the requirement that the derived objects be declared in the same unit.

See also

[Component visibility](#)

[Private](#)

[Public](#)

[Published](#)

[Automated](#)

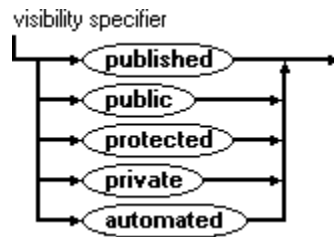
[Object-type scope](#)

Public

See also

[Standard directives](#)

Syntax



Description

The **public** directive is used within class type declarations.

Component identifiers declared in **public** component parts have no special restrictions on their scope.

See also

[Component visibility](#)

[Private](#)

[Protected](#)

[Published](#)

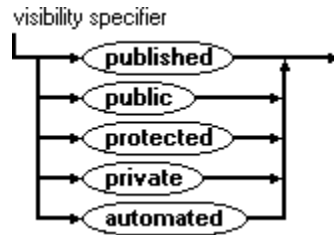
[Automated](#)

[Object-type scope](#)

Published

See also [Standard directives](#)

Syntax



Description

The **published** directive is used in object type declarations.

Declaring a part of an object as published generates run-time type information for that part, including it in the application's published interface.

Inside your application, a **published** part acts just like a **public** part. The only difference is that other applications can get information about those parts through the **published** interface.

The C++Builder Object Inspector uses the published interface of objects in the Component palette to determine the properties and events it displays.

See also

[Component visibility](#)

[Private](#)

[Protected](#)

[Public](#)

[Automated](#)

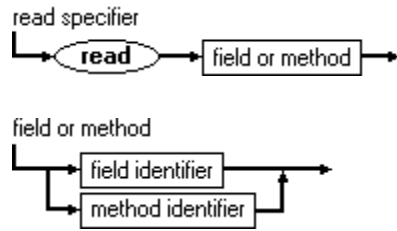
[Object-type scope](#)

Read

[See also](#)

[Example](#)

Syntax



Description

The **read** directive enables you to specify a routine or field that will get a value from a property.

See also
Write

Example

property Color: TColor **read** GetColor **write** SetColor;

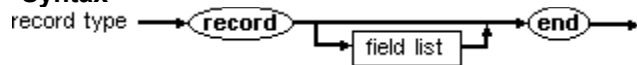
Record

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

A **record** contains components, or fields, that can be of different types.

Each field list separates identifiers with a comma, followed by a colon and a type.

You must name the field and assign a field type in the record-type declaration.

The variant part of the record type syntax diagram distributes memory space for more than one list of fields, letting you access information in more than one way. Each field list is a variant which overlays the same space in memory. Each variant is distinguished by a constant, and you can access all fields of all variants at all times.

The optional identifier, the tag field identifier, is the identifier of an additional fixed field--the tag field--of the record. The program uses the tag field's value to show which variant is currently active.

Accessing records

You can access the whole record or each field individually. To retrieve information from an individual field type the record name, a period, and then the field identifier. For example,

```
TDateRec.Year
```

If a record contains a subrecord, you can access it using qualifier.

Example

```
{ Record Type Definitions }  
type  
  TClass = (Num, Dat, Str);  
  TDate  = record  
    D, M, Y: Integer;  
end;  
  Facts = record  
    Name: string[10];  
    case Kind: TClass of  
      Num: (N: Real);  
      Dat: (D: TDate);  
      Str: (S: string);  
end;
```

See also

[Field and object component designators](#)

[Record-type constants](#)

[Record scope](#)

[With statements](#)

Register

[See also](#)

[Standard directives](#)

Description

The **register** directive specifies that a procedure or function uses the register calling convention for passing parameters. This is the default calling convention in this version of Object Pascal.

The register calling convention passes parameters from left to right, with parameters removed from the stack by the function.

The register convention uses up to three CPU registers to pass parameters, where the other conventions always pass all parameters on the stack. The register convention is by far the most efficient calling convention, since it often avoids the creation of a stack frame.

See also

[Calling conventions](#)

[cdecl](#)

[pascal](#)

[stdcall](#)

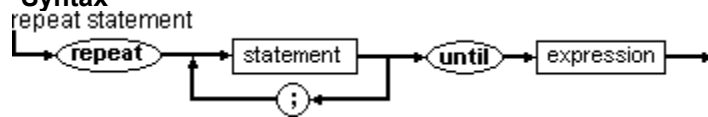
Repeat...Until

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

The statements between **repeat** and **until** are executed in sequence while the Boolean expression in the **until** statement evaluates to True.

Using this loop ensures that the sequence is executed at least once because the Boolean expression is evaluated after the execution of each sequence.

Example

```
{ Repeat Statements }  
repeat Ch := GetChar until Ch <> ' '  
repeat  
    Write('Enter value: ');  
    ReadLn(I);  
until (I >= 0) and (I <= '9');
```

See also

Loops

Resident

Standard directives

The **resident** standard directive is included in an exports clause.

When **resident** is used, the export information stays in memory when the dynamic-link library (DLL) is loaded.

The resident option reduces the time it takes Windows to look up a DLL entry by name.

If client programs that use the DLL are likely to import certain entries by name, they should be exported using the **resident** standard directive.

Self

Reserved words

Self is an implicit parameter passed whenever a method is called.

Self can be used as a pointer to the instance through which the method is being called.

It guarantees, among other things, that the correct virtual methods will be called for a particular object instance and that the correct instance data will be used by the object method.

Set

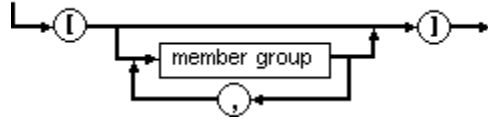
[See also](#)

[Example](#)

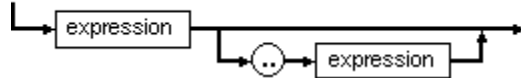
[Reserved words](#)

Syntax

set constructor



member group



Description

The reserved word **set** defines a collection of objects of the same ordinal type with no more than 256 possible values.

The ordinal values of the upper and lower bounds of the base type must be between 0 and 255.

A set constructor, which denotes a set-type value, is formed by writing expressions within brackets. Each expression denotes a value of the set.

The notation [] denotes the empty set, which is compatible with all set types.

Example

```
{ Set types }
  type
    Day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
    CharSet = set of Char;
    Digits = set of 0..9;
    Days = set of Day;
{ Set constructors }
['0'..'9', 'A'..'Z', 'a'..'z', '_']
[1, 5, I + 1 .. J - 1]
[Mon..Fri]
```

See also

Of

Set types

Set-type constants

Stdcall

See also [Standard directives](#)

Description

The **stdcall** directive specifies that a procedure or function uses the Windows standard calling convention for passing parameters.

The stdcall convention passes parameters from right to left, like the **cdecl** convention, but with parameters removed from the stack by the function.

The stdcall calling convention is used for calling Windows API routines.

See also

[Calling conventions](#)

[cdecl](#)

[pascal](#)

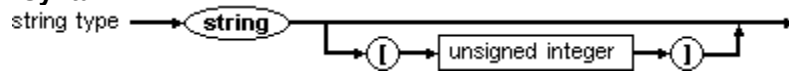
[register](#)

String

[See also](#)

[Reserved words](#)

Syntax



Description

The reserved word **string** is used to declare string type variables.

See also

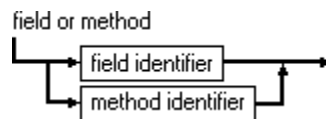
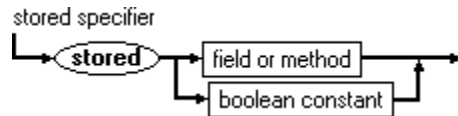
[Indexes](#)

[String types](#)

Stored

[Standard directives](#)

Syntax



Description

The **stored** directive controls whether or not a property is filed.

If present in a property definition, the **stored** directive must be followed by one of the following:

- A Boolean constant (True or False)
- The identifier of a field of type Boolean
- The identifier of a parameterless function method that returns a value of type Boolean

If a property definition does not include a **stored** specifier, the results are the same as if a **stored** True specifier had been included.

Threadvar

[See also](#) [Reserved words](#)

The **threadvar** reserved word is used to declare thread-local variables. Its syntax is identical to that of the **var** reserved word.

See also

[Var](#)

Try

[See also](#)

[Example](#)

[Reserved words](#)

The **try** reserved word is used to mark the first part of a protected block. There are two types of protected blocks:

- **try..except** block
- **try..finally** block

Syntax

⋮

The try...except Block

A block that handles exceptions is a **try..except** block.

Within the **try** part of the block, statements execute in the normal order unless an exception occurs, at which point execution jumps to the **except** part. If no exception occurs, the block ends without using the **except** or **else** parts.

The **except** part is a list of specific exceptions and responses to them, with each being an **on..do** statement. If none of the **on..do** statements applies to the current exception, the default exception handler in the **else** part executes. Once one handler (either a specific one or the default handler) deals with the exception, the block ends.

Execution does not resume within the block after an exception. In the example above, if Statement1 causes an exception, Statement2 never executes.

The try..finally Block

In order to ensure that resources allocated by your application are also released, you can protect resource allocations with a **try..finally** block.

The statements in the **finally** part of a **try..finally** block always execute, even if an exception occurs.

The statements in a **try..finally** block execute normally unless an exception occurs, at which point the statements in the **finally** part execute. Note that the **try..finally** block does not itself handle particular exceptions.

Example

The following code shows a protected resource. Closing the file in the **finally** part of the block ensures that the application always closes the file, even if an exception occurs.

```
var
    F: File;
begin
    Assign(F, 'SOMEFILE.EXT');
    Reset(F);
    try
        { statements that access file F }
    finally
        Close(F);
    end;
end;
```

See also

[Exception handling](#)

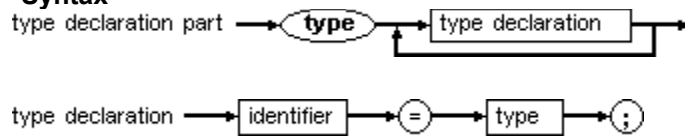
[Protecting resource allocations](#)

Type

[See also](#)

[Reserved words](#)

Syntax



Description

A **type** declaration specifies an identifier that denotes a type. A variable's **type** defines the set of values it can have and the operations that can be performed on it.

See also

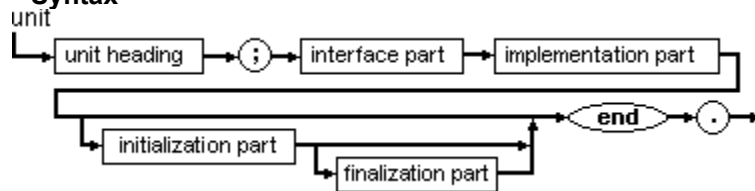
[Type declarations](#)

Unit

See also

[Reserved words](#)

Syntax



Description

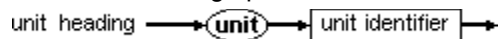
Units are the basis of modular programming. You use units to create libraries and to divide large programs into logically related modules.

These are the parts of a unit:

- **unit** heading
- interface part
- implementation part
- initialization part
- finalization part

Unit heading

The **unit** heading specifies the unit's name, which you use when referring to the unit in a uses clause.



The name must be unique: Two units with the same name cannot be used at the same time.

See also

[Circular unit references](#)

[Indirect unit references](#)

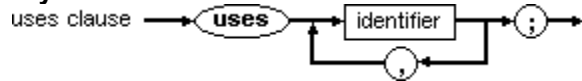
Uses

[Example](#)

[Reserved words](#)

The **uses** clause identifies all units used by the program.

Syntax



Description

Each identifier in a **uses** clause names a unit that has functions or procedures accessed by the current program or unit.

The System unit is always used automatically. System implements all low-level, run-time routines to support such features as file input and output (I/O), string handling, floating point, dynamic memory allocation, and others.

Apart from System, Object Pascal implements many standard units that aren't used automatically; you must include them in your **uses** clause.

The order of the units listed in the **uses** clause determines the order of their initialization.

To find the unit file containing a compiled unit, the compiler adds the file extension .DCU to the unit name listed in the **uses** clause.

The compiler searches for units in the current directory and in the directories specified in the Unit Directories list box on the Directories/Conditionals page of the Project Options dialog box.

Example

```
program MyProgram;  
uses SysUtils;
```

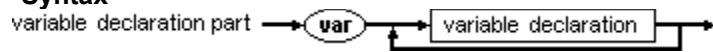
Var

[See also](#)

[Example](#)

[Reserved words](#)

Syntax



Description

A variable (**var**) declaration associates an identifier and a type with a location in memory where values of that type can be stored.

An absolute clause can be used to specify an absolute memory address.

The **var** reserved word is also used to declare variable parameters.

Example

```
{ Variable Declarations }  
var  
  X, Y, Z: real;  
  I, J, K: Integer;  
  Done, Error: Boolean;  
  Vector: array[1..10] of real;  
  Name: string[15];  
  InFile, OutFile: Text;  
  Letters: set of 'A'..'Z';
```

See also

[Global and local variables](#)

[Scope](#)

[Variable declarations](#)

Virtual

[See also](#)

[Standard directives](#)

Description

The **virtual** directive is used to declare a virtual method.

A **virtual** method is linked to its code at run time, by a process called late binding.

Declaring a method as **virtual** makes it possible for methods with the same name to be implemented in different ways within a hierarchy of object types.

To make a method virtual, follow its declaration in the object type with a semicolon, followed by the reserved word **virtual**.

See also

Objects

Self

Virtual methods

While

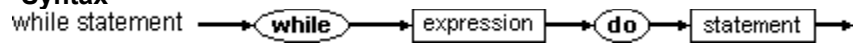
[See also](#)

[Example](#)

[Reserved words](#)

Syntax

while statement → **while** → expression → **do** → statement →



Description

A **while** statement controls the repeated execution of a singular or compound statement.

The statement after **do** executes as long as the Boolean expression is True.

The expression is evaluated before the statement is executed, so if the expression is False at the beginning, the statement is not executed.

See also

[Compound statements](#)

[Do \(reserved word\)](#)

[For \(reserved word\)](#)

[Loops](#)

[Repeat \(reserved word\)](#)

Example

```
{ while statements }  
  while Ch = ' ' do Ch := GetChar;
```

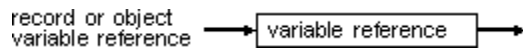
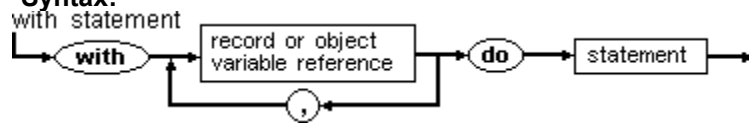
With

[See also](#)

[Example](#)

[Reserved words](#)

Syntax:



Description

The **with** statement is a shorthand method for referencing the fields of a record and the fields and methods of an object.

Within a **with** statement, the fields of one or more record variables can be referenced using only their field identifiers.

Within a **with** statement, each variable reference is first checked to see if it can be interpreted as a field of the record. If so, it is always interpreted as such, even if a variable with the same name is also accessible.

If the selection of a record variable involves indexing an array or dereferencing a pointer, these actions are executed once before the component statement is executed.

See also
Records

Example

type

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

end;

var OrderDate: TDate;

with OrderDate **do**

if Month = 12 **then**

begin

 Month := 1;

 Year := Year + 1

end

else

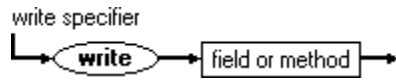
 Month := Month + 1;

Write

[See also](#)

[Example](#)

Syntax



.

Description

The **write** directive is a property access specifier that enables you specify a routine that will set the value of a property.

See also

[Read](#)

Type declarations

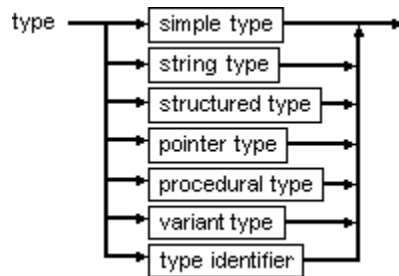
See also [Language definition](#)

When you declare a variable, you must state its type. Types can be either predefined or user-defined. User-defined types are declared in the type declaration part of a program or unit.

A variable's type defines the set of values it can have and the operations that can be performed on it.

The scope of a type declaration is within the block in which it was declared.

A type identifier's scope does not include itself, with the exception of pointer types.



There are six major classes of types:

1. Simple types define ordered sets of values.
2. String types define a sequence of characters with a dynamic length attribute and a constant size attribute.
3. Structured types define a structure that can hold more than one value.
4. Pointer types define a set of values that point to variables of a specified type.
5. Procedural types allow procedures and functions to be treated as objects.
6. Variant types allow variables to assume values of different types.

See also

Scope

Type compatibility

Fundamental and generic types

Types

Object Pascal's predefined types are divided into two categories:

- Fundamental types
- Generic types

The range and format of fundamental types is independent of the underlying CPU and operating system and does not change across different implementations of Object Pascal.

The range and format of generic types depends on the underlying CPU and operating system.

There are currently three classes of predefined types that distinguish between fundamental and generic types:

- Integer types
- Character types
- String types

For all other classes, you should regard the predefined types as fundamental.

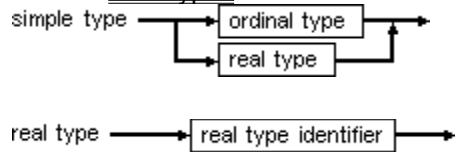
Applications should use the generic formats whenever possible, since they generally result in the best performance for the underlying CPU and operating system. The fundamental types should be used only when the actual range and / or storage format matters to the application.

Simple types

Types

Simple types define ordered sets of values. There are two base classes for simple types:

- Ordinal types
- Real types



A real type identifier is one of the standard identifiers: Real, Single, Double, Extended, or Comp.

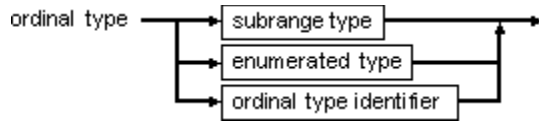
Comparing simple types

When comparing simple types, the operands must be of compatible types; however, if one operand is of a real type, the other can be an integer type.

Ordinal types

See also [Types](#)

Ordinal types are a subset of simple types and contain a finite number of elements.



C++Builder has twelve predefined ordinal types and two user-defined types. The predefined types are:

Integer Shortint SmallInt Longint
Byte Word Cardinal Char
Boolean ByteBool WordBool LongBool

The user-defined ordinal types are:

- enumerated types
- subrange types

All values of an ordinal type are an ordered set, and each value is represented by its index position within that set. Except for integer type values, the first element of every ordinal type has the position index 0, the next is 1, and so on. The position index of an integer type value is the value itself.

You can use the following standard functions with ordinal types:

Function	What it does
Ord	Returns the element's numerical ordering within the set.
Pred	Returns the predecessor of the value. If the value in question is the first value in the ordinal type and if range checking is enabled <u>{SR+}</u> , Pred produces a run-time error that you can handle by using exceptions.
Succ	Returns the successor of the value. If the value in question is the last value in the ordinal type and if range checking is enabled <u>{SR+}</u> , Succ produces a run-time error that you can handle by using exceptions.
Low	Determines the lowest value in the range of the given ordinal type.
High	Determines the highest value in the range of the given ordinal type.

See also

[Exception handling](#)

[Type compatibility](#)

[Variable typecasting](#)

[Value typecasting](#)

Integer types

See also [Ordinal types](#)

Object Pascal's predefined integer types are divided fundamental and generic types.

Applications should use the generic integer formats whenever possible, since they generally result in the best performance for the underlying CPU and operating system. The fundamental integer types should be used only when the actual range and / or storage format matters to the application.

Fundamental types

The fundamental integer types are:

Type	Range	Format
Shortint	-128 .. 127	Signed 8-bit
Smallint	-32768 .. 32767	Signed 16-bit
Longint	-2147483648 .. 2147483647	Signed 32-bit
Byte	0 .. 255	Unsigned 8-bit
Word	0 .. 65535	Unsigned 16-bit

The range and format of the fundamental types are independent of the underlying CPU and operating system and does not change across different implementations of Object Pascal.

Generic types

The generic integer types are Integer and Cardinal. The Integer type represents a generic signed integer, and the Cardinal type represents a generic unsigned integer. The actual ranges and storage formats of the generic types vary across different implementations of Object Pascal, but are generally the ones that result in the most efficient integer operations for the underlying CPU and operating system.

Type	Range	Format
Integer	-32768 .. 32767	Signed 16-bit
Integer	-2147483648 .. 2147483647	Signed 32-bit
Cardinal	0 .. 65535	Unsigned 16-bit
Cardinal	0 .. 2147483647	Unsigned 32-bit

Arithmetic operations

Arithmetic operations with integer-type operands use 8-bit, 16-bit, or 32-bit precision, according to the following rules:

- An integer constant takes the integer type with the smallest range that includes the value of the integer constant.
- Both operands in a binary operation, convert to the integer type with the smallest range that includes all possible values of both types. The resulting type of the expression is the common type.
- The expression on the right of an assignment statement evaluates independently from the size or type of the variable on the left.
- Bytes convert to an intermediate word operand compatible with both Integer and Word before the statement evaluates.

Note: You can explicitly convert an integer-type value to another integer type through typecasting.

See also

[Type compatibility](#)

[Value typecasting](#)

[Variable typecasting](#)

[Fundamental and generic types](#)

Boolean types

See also [Ordinal types](#)

There are four predefined Boolean types. The Boolean type declares variables that will evaluate to either False or True.

Type	Memory
Boolean	1 byte
ByteBool	1 byte
WordBool	two bytes (one word)
LongBool	four bytes (two words)

The most common use of a Boolean expression is with relational operators and conditional statements. Since Boolean types are enumerated types, the following relationships apply:

- $\text{False} < \text{True}$
- $\text{Ord}(\text{False}) = 0$
- $\text{Ord}(\text{True}) = 1$
- $\text{Succ}(\text{False}) = \text{True}$
- $\text{Pred}(\text{True}) = \text{False}$

Boolean is the preferred type and uses less memory; ByteBool, WordBool, and LongBool provide compatibility with other languages and the Windows environment.

Unlike Boolean variables, which can only assume the values 0 (False) or 1 (True), ByteBool, WordBool, and LongBool can assume other ordinal values where 0 is False and any nonzero value is True. Whenever a ByteBool, WordBool, or LongBool value is used in a context where a Boolean value is expected, the compiler will automatically generate code that converts any nonzero value to the value True.

See also

[Boolean expressions](#)

[Boolean operators](#)

[Conditional statements](#)

[Relational operators](#)

[Type compatibility](#)

Character types

See also [Ordinal types](#)

Object Pascal defines two fundamental character types and a generic character type.

The fundamental character types are

- AnsiChar Byte-sized characters, ordered according to the extended ANSI character set.
- WideChar Word-sized characters, ordered according to the Unicode character set. The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is Char.

In the current implementation of Object Pascal, Char corresponds to the fundamental type AnsiChar, but implementations for other CPUs and operating systems might define Char to be WideChar. When writing code that might need to handle characters of either size, use the standard function SizeOf instead of a hard-coded constant for character size.

The function call Ord(Ch), where Ch is any character-type value, returns Ch's ordinality.

A string constant of length 1 can be represented by a constant character value. The Chr function can convert an Integer value into a character with the corresponding ordinality.

See also

[Type compatibility](#)

[Fundamental and generic types](#)

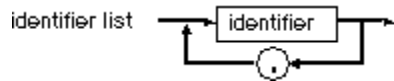
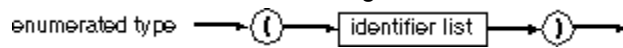
Enumerated types

[See also](#)

[Example](#)

[Ordinal types](#)

Enumerated types assign sequential values to elements in an identifier list. The first element gets the value 0; the second element gets 1, and so on.



The compiler recognizes the enumerated type name as the type for all the entire identifier list.

An identifier's ordinality is determined by its position within the identifier list in which it is declared.

Use the Succ and Pred functions to cycle forward or backward through the elements of the identifier list.

When the Ord function is applied to an enumerated type's value, Ord returns an integer that shows where the value falls with respect to the other values of the enumerated type.

See also

[Boolean types](#)

[Type compatibility](#)

[Type declarations](#)

Enumerated type example

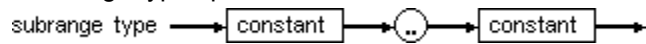
type

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

Subrange types

[Example](#) [Ordinal types](#)

A subrange type is a range of values from an ordinal type called the host type. The definition of a subrange type specifies the smallest value and the largest value in the subrange.



Both constants must be of the same ordinal type.

A variable of a subrange type has all the properties of the host type, but its run-time value must be within the specified range.

One syntactic ambiguity occurs in constant expressions, since the compiler defines a type definition starting with a parenthesis as an enumerated type. There are two possible solutions to this problem:

- Reorganize the first subrange expression so that it does not start with a parenthesis.
- Set a constant equal to the value of the expression and use that constant in the type definition.

The [\\$R compiler directive](#) controls range checking of subrange types.

Subrange type examples

These are examples of subrange types:

0..99

-128..127

Club..Heart

The following example is one possible solution to the constant expression problem:

type

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

Real types

[See also](#) [Types](#)

A real type is a subset of real numbers, which you can represent in floating-point notation with a fixed number of digits.

There are six kinds of real types; they differ in the range, precision of values and in size.

Type	Range	Significant digits	Size in bytes
Real	$2.9 * 10e-39 .. 1.7 * 10e38$	11-12	6
Single	$1.5 * 10e-45 .. 3.4 * 10e38$	7-8	4
Double	$5.0 * 10e-324 .. 1.7 * 10e308$	15-16	8
Extended	$3.4 * 10e-4932 .. 1.1 * 10e4932$	19-20	10
Comp	$-2e63+1 .. 2e63-1$	19-20	8
Currency	$922337203685477.5808..922337203685477.5807$	19-20	8

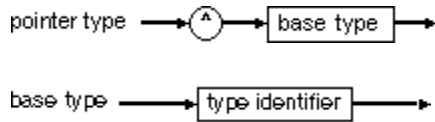
The Comp (computational) type holds only integral values within $-2e63+1$ to $2e63-1$, which is approximately $-9.2 * 10e18$ to $9.2 * 10e18$.

The Currency type is a fixed-point data type suitable for monetary calculations. It is stored as a scaled 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Pointer types

[See also](#) [Example](#) [Types](#)

A pointer type is a value that points to variables of a base type. A pointer-type variable contains the memory address of a variable.



If the base type is an undeclared identifier, you must declare it in the same type declaration part as the pointer type.

You can assign a value to a pointer variable using the following:

Procedure/Function	What it does
New	Allocates a new memory area in the application heap for a dynamic variable and stores the address of that area in the pointer variable
<u>@ operator</u>	Directs the pointer variable to the memory area containing any existing variable or procedure or function entry point, including variables that already have identifiers
GetMem	Creates a new dynamic variable of a specified size, and puts the address of the block in the pointer variable

The reserved word **nil** denotes a pointer-valued constant that does not point to anything.

Comparing pointers

The operators = and <> can be used on compatible pointer-type operands. Two pointers are equal only if they point to the same object.

Compatibility note

C++Builder allows the use of undeclared identifiers in the declaration of pointer types only in the following context:

type

```
PointerType = ^UndefinedType;
```

where UndefinedType is defined later in the same type-declaration block.

See also

[Pointer-type constants](#)

[Pointers and dynamic variables](#)

[Type compatibility](#)

[Type Pointer](#)

[Type PChar](#)

[Value typecasting](#)

Type Pointer

[See also](#)

The predefined type Pointer is an untyped pointer.

You cannot dereference variables of type Pointer; writing the pointer symbol ^ after such a variable is an error.

You can dereference generic pointers through typecasting.

Values of type Pointer are compatible with all other pointer types.

See also

[Pointer Types](#)

[Pointers and Dynamic Variables](#)

[Type Compatibility](#)

[Type PChar](#)

[Variable Typecasting](#)

Character-pointer types

[See also](#)

Object Pascal defines two fundamental character-pointer types and a generic character-pointer type.

Character-pointer types are simply pointers to character types, but Object Pascal supports a set of extended syntax rules to facilitate handling of null-terminated strings using character-pointer types.

The fundamental character-pointer types are

PAnsiChar a pointer to a null-terminated string of characters of type AnsiChar

PWideChar a pointer to a null-terminated string of characters of type PWideChar

The generic character-pointer type is

PChar a pointer to a null-terminated string of characters of type Char

The System unit declares the pointer-character types as follows:

type

```
PAnsiChar = ^AnsiChar;
```

```
PWideChar = ^WideChar;
```

```
PChar = PAnsiChar;
```

The fundamental character-pointer types are pointers to the fundamental character types (or to null-terminated strings of such characters), and the generic character-pointer type is a pointer to the generic character type.

Comparing character pointers

Object Pascal allows the <, >, <=, or >= operators to be applied to character-pointer values. These relational tests assume that the two pointers being compared point within the same character array, and for that reason, the operators compare only the offset parts of the two pointer values.

If the two character pointers point to different character arrays, the result is undefined.

See also

[Character-pointer operators](#)

[Null-terminated strings](#)

[Pointer types](#)

[Relational operators](#)

[Type compatibility](#)

[Type Pointer](#)

Pointer type example

```
{ Pointer Type Declaration }
```

```
type
```

```
BytePtr = ^Byte;
```

```
WordPtr = ^Word;
```

```
IdentPtr = ^IdentRec;
```

```
IdentRec = record
```

```
    Ident: string[15];
```

```
    RefCount: Word;
```

```
    Next: IdentPtr;
```

```
end;
```

Standard Pointers

[See also](#)

When you define a structure or a data type in Pascal, you should also define a pointer to that data type. Many advanced programming techniques, such as linked lists of dynamically allocated records, may require you use the pointer instead of the variable itself.

When you a variable the compiler generates the necessary code to initialize and finalize these special types automatically. When you write code that allocates and frees the pointers to dynamic structures like AnsiString, and Variant, they require special initialization and finalization.

Some pointers are listed in the following table.

Pointer	Points to
PAnsiString	Points to an AnsiString variable.
PByteArray	Points to a variable of type TByteArray. Often used to typecast dynamically allocated blocks of memory for array access.
PCurrency	Points to a variable of type Currency.
PExtended	Points to a variable of type Extended.
PTextBuf	Points to a variable of type TextBuf. TextBuf is the internal buffer used in the TTextRec text file record.
PVarRec	Points to a variable of type TVarRec.
PVariant	Points to a variable of type Variant.
PWordArray	Points to a variable of type TWordArray. Often used to typecast dynamically allocated memory blocks for use as an array of word-sized (2 byte unsigned) values.

See also

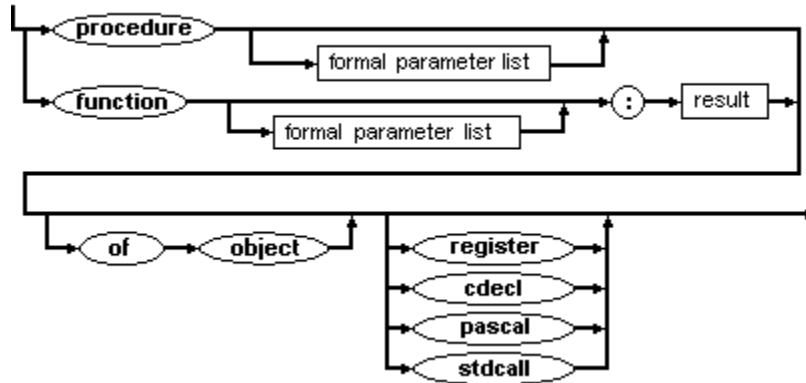
[Character-pointer operators](#)

Procedural types

See also [Example](#) [Types](#)

Procedural types enable you to treat procedures and functions as entities that can be assigned to variables and passed as parameters.

procedural type



The syntax for a procedural type declaration is the same as that of a procedure or function header, except that the identifier after the **procedure** or **function** keyword is omitted. A procedural-type declaration can optionally specify a calling convention. The default calling convention is **register**.

There are two categories of procedural types:

- Global procedure pointers
- Method pointers

You cannot declare functions that return procedural-type values. However, you can return the address of a procedure or function using a function result of type Pointer and then typecast it to a procedural type.

A function result must be a one of the following types:

- string
- real
- integer
- char
- Boolean
- Pointer
- user-defined enumeration

Procedural-type compatibility

In order for procedural types to be compatible the following conditions apply:

- Both types must use the same calling convention.
- Both types must have the same number of parameters.
- Parameters in corresponding positions must be of identical types.
- The result types of functions must be identical.

The value **nil** is compatible with any procedural type.

Note: Parameter names have no significance when determining procedural-type compatibility.

Global procedure pointer types and method pointers are mutually incompatible.

See also

[Procedural-type constants](#)

[Procedural values](#)

[Type pointer](#)

[Variable typecasting](#)

Procedural type examples

type

```
Proc = procedure;  
SwapProc = procedure(var X, Y: Integer);  
StrProc = procedure(S: string);  
MathFunc = function(X: Double): Double;  
DeviceFunc = function(var F: Text): Integer;  
MaxFunc = function(A, B: Double; F: MathFunc): Double;
```

var

```
P: SwapProc;  
F: MathFunc;
```

```
procedure Swap(var A, B: Integer);
```

var

```
Temp: Integer;
```

begin

```
Temp := A;  
A := B;  
B := Temp;
```

end;

```
function Tan(Angle: Double);
```

begin

```
Tan := Sin(Angle) / Cos(Angle);
```

end;

```
{ You can assign the variables P and F to the following values }
```

```
P := Swap;
```

```
F := Tan;
```

```
{ The following calls using P and F are now legal }
```

```
P(I, J);      { Equivalent to Swap(I, J) }
```

```
X := F(X);    { Equivalent to X := Tan(X) }
```

Global procedure pointers

[See also](#)

[Example](#)

A global procedure pointer is a procedural type declared without the **of object** clause.

Global procedure pointers can reference a global procedure or function, and is encoded as a pointer that stores the address of a global procedure or function.

See also

[Procedural types](#)

Example

type

```
TProcedure = procedure;  
TStrProc = procedure(const S: string);  
TMathFunc = function(X: Double): Double;
```


Procedural values

[See also](#) [Example](#)

You can assign a procedural value to a procedural-type variable.

Procedural values can be one of following:

- The value **nil**
- A variable reference of a procedural type
- A procedure or function identifier
- A method designator

When you declare a procedure or function as a procedural value, it is considered a constant declaration.

Using a procedural variable that has the value **nil** in a procedure statement or a function call results in an error. **nil** indicates an unassigned procedural variable. Procedure statements or function calls involving a **nil** procedural variable should use the following test. The **@** operator indicates that P is being examined rather than being called.

```
if @P <> nil then P(I, J);
```

The following types of procedures and functions cannot be used as procedural values:

- standard
- nested
- methods
- inline

Although you cannot directly use standard procedures and functions, there is a workaround. To use standard procedures or functions as a procedural values, you must declare a new function or procedure that calls the standard procedure or function in its main body.

See also

[Procedural types](#)

[Type compatibility](#)

[Using procedural types in expressions](#)

String types

[See also](#)

[Example](#)

[Types](#)

C++Builder supports two types of strings:

- [short strings](#)
- [long strings](#)

Short strings and long strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions.

The short string type represents a statically allocated string with maximum length between 1 and 255 characters, and a dynamic length between 0 and the maximum length.

The long string type represents a dynamically allocated string with a maximum length limited only by available memory. For new applications, it is recommended that you use the long string type.

Note A new compiler directive, `$H`, controls whether the reserved word **string** represents a short string or a long string. In the default state, `{ $H+ }`, **string** represents a long string, and using the **string** keyword is the same as using the predefined identifier `AnsiString`. In the `{ $H- }` state, **string** represents a short string with a maximum length of 255 characters, and using the **string** keyword is the same as using the predefined identifier `ShortString`.

Two consecutive single quotes are used to indicate a single quote in a string.

Example

```
{ String Type Definitions }  
const  
    LineLen = 79;  
type  
    Name = string[25];  
    Line = string[LineLen];
```

See also

[String \(reserved word\)](#)

[String operators](#)

[String-type constants](#)

[Type compatibility](#)

Short string types

See also [String types](#)

The declaration of a short-string-type specifies a maximum length between 1 and 255 characters. Variables of a short-string-type can contain strings with a dynamic length between 0 and the declared maximum length.

The predefined type `ShortString` denotes a short-string-type with a maximum length of 255 characters. The number of bytes of storage occupied by a short-string-type variable is the maximum length of the short-string-type plus one.

When assigning a string value to a short string variable, the string value is truncated if it is longer than the declared maximum length of the short string variable.

You can index a short string variable with a single index expression, whose value must be in the range $0..N$, where N is the declared maximum length of the short string. The type of a character accessed through indexing of a short string is `Char`. The index of the first character in a string is 1. The element at index 0 contains the dynamic length of the string, and for a short string, `Length(S)` is the same as `Ord(S[0])`. Assigning a value to the zeroth element of a short string alters the dynamic length of the string, but the compiler doesn't check whether the value is less than the declared maximum length of the string. It is possible to index a short string beyond its current dynamic length. The character values read in that case are undefined and assignments to character positions beyond the current length do not affect the actual value of the short string variable.

The `Low` and `High` standard functions can be applied to a short-string-type variable. In this case, `Low` returns zero, and `High` returns the declared maximum length of the short string.

See also

[String types](#)

[Long string types](#)

[String \(reserved word\)](#)

Long string types

See also [String types](#)

The long-string-type is denoted by the reserved word **string** and by the predefined identifier `AnsiString`.

Note If you change the state of the **\$H** compiler switch to **{\$H-}**, the reserved word **string** denotes a short-string with a maximum length of 255 characters. However, the predefined identifier `AnsiString` always denotes the long-string-type.

Long strings are dynamically allocated and have no declared maximum length. The theoretical maximum length of a long string value is 2GB (two gigabytes). In practice this means that the maximum length of a long string value is limited only by the amount of memory available to an application.

Management of the dynamically allocated memory associated with a long string variable is entirely automatic and requires no additional user code. The automatic management of long strings has the following characteristics:

- 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 non-empty 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.
- Long strings are reference counted. When a long string variable is assigned a new value, the reference count of its previous value (if non-empty) is decremented, and the reference count of its new value (if non-empty) is incremented. When the reference count of a string value reaches zero, the dynamically allocated memory that contains the string value is deallocated. Reference counting dramatically reduces string-data copying and memory consumption. The only data that is copied in a long string assignment is the 32-bit string value pointer, and any number of long string variables can reference the same value without consuming additional memory. For this reason, long string assignments typically execute faster than short string assignments.
- When indexing is used to change the value of a single character in a long string variable, a copy of the string value is first created if the string value's reference count is greater than one. This is known as copy-on-write semantics, and guarantees that the modification does not also modify other long string variables that reference the string value.
- Long string variables are *always* initialized to be empty when they are first created. This is true whether a long string variable is global, local, or part of a structure such as an array, record, or object.
- When long string variables go out of scope (such as upon exiting a function with local long string variables, or upon destroying an object that contains long string fields), the reference count of their values are automatically decremented. For a function this is true even if the function is exited because of an exception.
- You can index a non-empty long string variable with a single index expression, whose value must be in the range 1..N, where N is the dynamic length of the long string. The type of a character accessed through indexing of a long string is `Char`. The index of the first character in a long string is 1. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the `Length` standard function, and to set the length of a long string you must use the `SetLength` standard procedure.

While it is possible to assign to a character position obtained by indexing a long string, it is not possible to pass such a character as a **var** parameter.

When declaring long string fields in a record type, all such fields must reside in the non-variant part of the record type. In other words, long string fields or fields of a type that contains long strings are not allowed in a variant part of a record type.

Long strings and null-terminated strings

Dynamic memory allocated for a long string value is automatically terminated by a null character (the null character is not part of the string, but rather is automatically stored right after the last character in the string). Because of the automatic null character termination, it is possible to directly typecast a long string value to a `PChar` value. The syntax of such a typecast is `PChar(S)`, where `S` is a long string expression. A `PChar` typecast returns a pointer to the first character of the long string value, and is

guaranteed to return a pointer to a null terminated string even if the string expression is empty.

The following example shows how PChar typecasts can be used to pass long string values to a function that expects null-terminated string parameters. Caption and Message are long string variables, and MessageBox is a Win32 API function defined in the Windows interface unit.

```
Caption := 'Hello world';  
Message := 'This is a test of long strings';  
MessageBox(0, PChar(Message), PChar(Caption), MB_OK);
```

It is also possible to typecast a long string to an untyped pointer, using the syntax Pointer(S), where S is a long string expression. A Pointer typecast returns the address of the first character of the long string value. Unlike a PChar typecast, a Pointer typecast returns **nil** if the string expression is empty.

The lifetime of a pointer returned by a PChar or Pointer typecast depends on the argument of the typecast. If the argument is a long string expression, the pointer remains valid only within the statement in which the typecast is performed. This essentially limits such a typecast to use only in parameter expressions. If the argument is a long string variable, the pointer remains valid until a new value is assigned to the variable, or until the variable goes out of scope.

In general, the null-terminated string referenced by the pointer returned by a PChar or Pointer typecast should be considered read-only.

It is possible to use the pointer returned by a PChar or Pointer typecast to modify a corresponding long string, but it is safe to do so only in certain situations. Given a typecast of the form PChar(S) or Pointer(S), the null-terminated string referenced by the result of the typecast can be modified only if all of the following requirements are satisfied:

- S is a long string variable (not an expression).
- The value of S is not empty. In other words, S must contain a string with a length greater than zero.
- The value of S is unique, that is the long string value has a reference count of one. Following a call to the SetLength, SetString, and UniqueString standard procedures, a long string variable's value is unique, and it is guaranteed to remain unique as long as the string variable is not referenced in a string expression.
- S has not been modified and has not gone out of scope since it was typecast.
- The characters modified all lie within the length of the string. In other words, when indexing the returned PChar value, index values must be between 0 and Length(S) – 1.

See also

[String types](#)

[Short string types](#)

[String \(reserved word\)](#)

[Null-terminated strings](#)

Using strings

See also [String types](#)

The ordering between any two string values is defined by the ordering relationship of the character values in corresponding positions. In two strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a higher or greater-than value. For example, 'BA' is greater than 'A'. Zero length strings are equal only to other zero length strings, and they hold the least string values.

Operators for the string types are described in "String operator" and "Relational operators" in Chapter 5.

It is possible to mix short and long string types in assignments and expressions, but strings passed as **var** parameters must be of the appropriate type. In other words, it is not possible to pass a short string as a **var** parameter to a procedure or function that expects a long string, and vice versa.

A short string can be explicitly converted to a long string using a typecast of the form `AnsiString(S)`, where S is a short string expression. Also, in the default **{\$H+}** state, a typecast of the form **string(S)**, will convert a short string expression to a long string.

A long string can be explicitly converted to a short string using a typecast of the form `ShortString(S)`, where S is a long string expression. Also, in the **{\$H-}** state, a typecast of the form **string(S)**, will convert a long string expression to a short string.

See also

[Long string types](#)

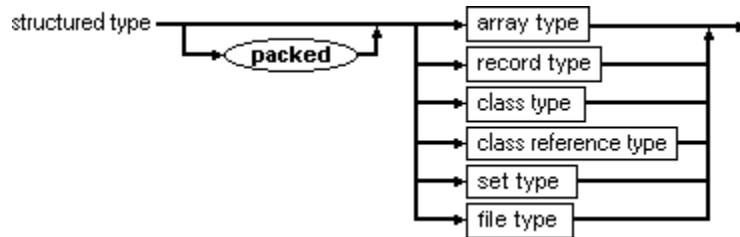
[Short string types](#)

[String \(reserved word\)](#)

Structured types

See also [Types](#)

A structured type holds more than one value. The components of structured types can be manipulated individually or as a whole, and can themselves be structured types. There is no limit to the number of such nested structures.



C++Builder's structured types are:

- [array types](#)
- [file types](#)
- [class types](#)
- [class reference types](#)
- [record types](#)
- [set types](#)

A structured type can have nested levels.

Class types and class reference types are the cornerstones of object oriented programming in Object Pascal.

The reserved word **packed** in a structured type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a variable of this type. By default Object Pascal aligns components of structured types on word or double-word boundaries for faster access. Adding **packed** to a structured type's declaration overrides such alignment for that type.

See also

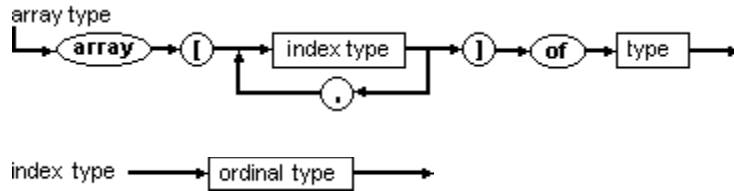
[Structured-type constants](#)

[Type compatibility](#)

Array types

See also [Example](#) [Structured types](#)

Arrays are one dimensional or multidimensional containers that hold multiple variables of the same type. Each variable of the array can be referred to by the array name and the its index enclosed in brackets.



To specify an array type you must give the compiler two pieces of information:

- The ordinal index type of an array that specifies the number of elements.
- The base type.

The number of elements in an array is the product of the number of values in each index type.

To access the elements of the array, add brackets and an index value to the array identifier. The following statement accesses the third element in the array:

```
array[3];
```

Use the standard functions Low and High with an array-type identifier or a variable reference of an array to return the low and high bounds of the index type of the array.

Multidimensional array types

If an array's component type is also an array, you can treat the result as an array of arrays or as a single multidimensional array. For example,

```
array[Boolean] of array[1..10] of array[Size] of Real
```

is interpreted the same way by the compiler as

```
array[Boolean,1..10,Size] of Real
```

When declaring multidimensional arrays the total number of elements in the array is the product of the number of values in each index type.

Zero-based arrays

Zero-based arrays are declared by assigning the first element in the array declaration an index of zero. For example,

```
array[0..5] of Char
```

Use zero-based character arrays to store null-terminated strings. A zero-based character array is compatible with a PChar value.

Array example

Here is a declaration of ;

```
array[1..100] of Real { declares an array that can hold 100 elements of  
type real }
```


See also

[Array \(reserved word\)](#)

[Array-type constants](#)

[Indexes](#)

[Null-terminated strings](#)

[Open-array parameters](#)

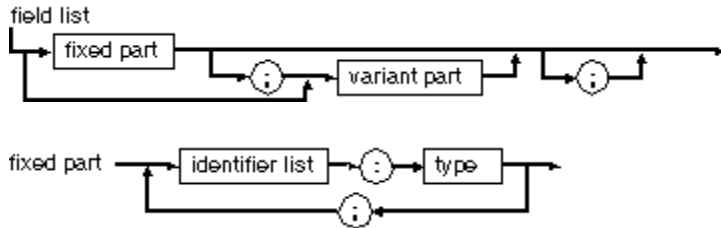
[Type compatibility](#)

[Variable reference](#)

Record types

See also [Example](#) [Structured types](#)

A record type is a collection of fields that can be of different types.

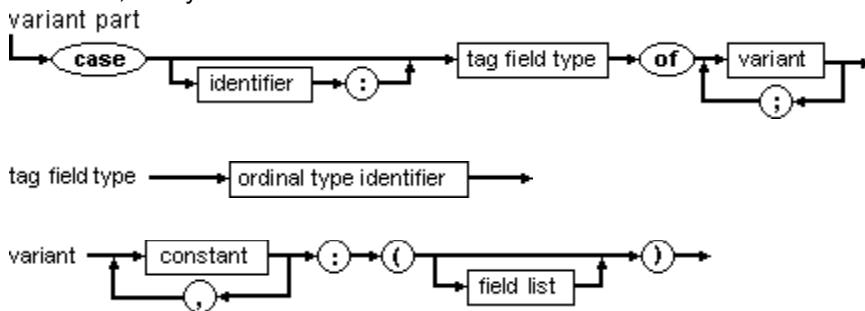


The field name and field type must be assigned in the record-type declaration.

The fixed part of a record type sets out the list of fixed fields, giving an identifier and type for each. Each field contains information that is always retrieved in the same way.

The variant part of a record type distributes memory space for more than one list of fields, letting you access information in more than one way.

Each field list is a variant that overlays the same space in memory. Each variant is distinguished by a constant, and you can access all fields of all variants at all times.



Each variant is identified by at least one constant. All constants must be unique and of an ordinal type compatible with the tag field type.

The optional tag field identifier, is the identifier for an additional fixed field--the tag field--of the record. The program uses the value of the tag field to show the active variant.

Without a tag field, the program selects a variant by another criterion.

Note Fields in the variant part of a record cannot be of a long string type or of the type Variant.

Likewise, fields in the variant part of a record cannot be of a structured type that contains long string or Variant components.

Accessing records

You can access the whole record or each field individually. To access an individual field, type the record name, a period, and then the field identifier. For example,

```
TDateRec.Year
```

To access an entire record, use the **with** statement.

See also

[Record scope](#)

[Type compatibility](#)

[With statement](#)

Record type examples

type

```
TDateRec = record  
  Year: Integer;  
  Month: 1..12;  
  Day: 1..31;  
end;
```

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: TDate;  
           ExitDate: TDate);  
end;
```

Class types

See also [Example](#) [Structured types](#)

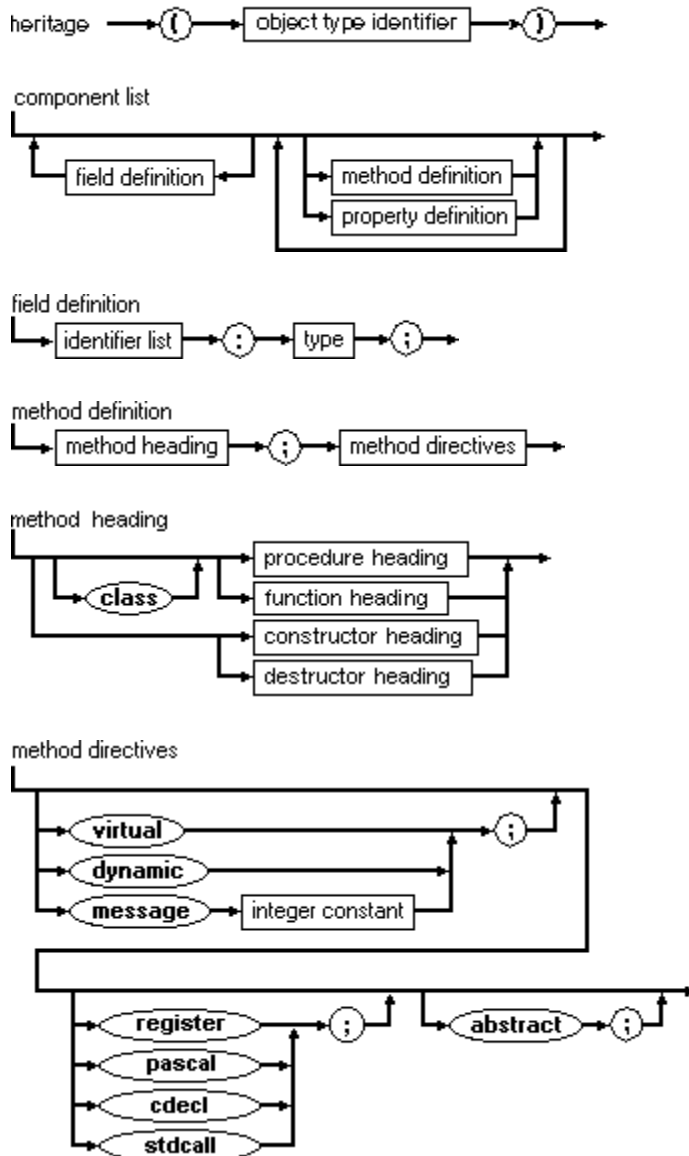
A class type is a structure consisting of a fixed number components.

Possible components of a class are

- fields
- methods
- properties

Unlike other types, a class type can be declared only in a type declaration part in the outermost scope of a program or unit. Therefore, a class type cannot be declared in a variable declaration part or within a procedure, function, or method block.

A class type is declared using the reserved word **class** and defines the contents of a class. Classes are also called "object types." The two terms are interchangeable.



You must globally declare a class type. A class type cannot be declared in a variable declaration part or within a procedure, function, or method block.

The component type of a file cannot be a class type, or any structured type with a class-type component.

Inheritance

A class type can inherit components from another class type. The inheriting class is a descendant and the class inherited from is its ancestor.

Inheritance is transitive; for example, if T3 inherits from T2, and T2 inherits from T1, then T3 also inherits from T1. The domain of a class type consists of itself and all its descendants.

A descendant class implicitly contains all the components defined by its ancestor classes. A descendant class can add new components to those it inherits. However, it cannot remove the definition of a component defined in an ancestor class.

The predefined class type TObject is the ultimate ancestor of all class types. If the declaration of a class type does not specify an ancestor type (that is, if the heritage part of the class declaration is omitted), the class type will be derived from TObject. TObject is declared by the System unit, and defines a number of methods that apply to all classes.

Class-type compatibility

A class type is assignment-compatible with any ancestor object type; therefore, during program execution, a class type variable can reference an instance of that type or an instance of any descendant type. For example, given the declarations

type

```
TFigure = class
:
end;
TRectangle = class(TFigure)
:
end;
TRoundRect = class(TRectangle)
:
end;
TEllipse = class(TFigure)
:
end;
```

a value of type TRectangle can be assigned to variables of type TRectangle, TFigure, and TObject, and during execution of a program, a variable of type TFigure might be either **nil** or reference an instance of TFigure, TRectangle, TRoundRect, TEllipse, or any other instance of a descendant of TFigure.

See also

[Class \(reserved word\)](#)

[Component scope](#)

[Component visibility](#)

[Instantiating objects](#)

[Methods](#)

Class type example

The following example declares the class TField

```
TField = class
private
  X, Y, Len: Integer;
  FName: string;
public
  constructor Copy(F: TField);
  constructor Create(FX, FY, FLen: Integer; FName: string);
  destructor destroy; override;
  procedure Display; virtual;
  procedure Edit; dynamic;
protected
  function GetStr: string; virtual;
  function PutStr(S: string): Boolean; virtual;
private
  procedure DisplayStr(X, Y: Integer; S: string);
public
  property Name: String read GetStr write Buffer;
end;

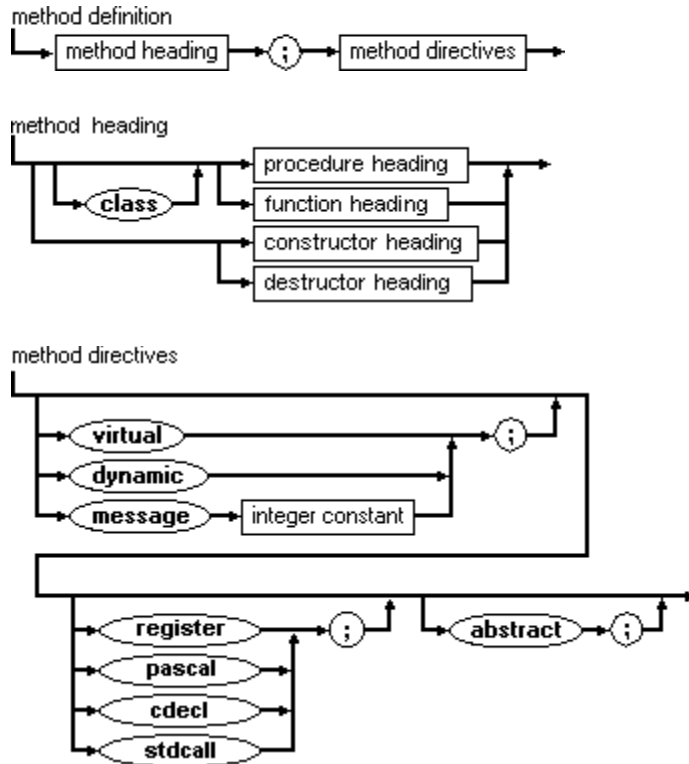
TStrField = class(TField)
private
  Value: PString;
public
  constructor Create(FX, FY, FLen: Integer; FName: string);
  destructor Destroy; override;
protected
  function GetStr: string; override;
  function PutStr(S: string): Boolean; override;
end;

TNumField = class(TField)
private
  Value, Min, Max: Longint;
public
  constructor Create(FX, FY, FLen: Integer; FName: string;
    FMin, FMax: Longint);
  function GetStr: string; override;
  function PutStr(S: string): Boolean; override;
  function Get: Longint;
  procedure Put(N: Longint);
end;
```


Methods

See also [Class types](#)

A method is a procedure or function declared inside an object-type declaration that performs an operation on an object.



Methods declared in an object type are by default static. When a static method is called, the declared (compile-time) type of the class or object used in the method call determines which method implementation to activate.

Methods can access the object's data fields without having them passed to it as parameters.

The declaration inside the object-type declaration corresponds to a forward declaration of that method.

The body of the method is defined outside the object type declaration but within the same scope. Its header must contain the name of the object type it is bound to., like this:

```

procedure ObjectType.Method(Param1, Param2: Integer);
begin
  ...
end; (* Method *)
  
```

Within the implementation of a method, the identifier `Self` represents an implicit parameter that references the object for which the method was invoked.

By default methods are static; however, they can also be any of the following types:

Virtual methods

Class methods

Constructors and destructors are special methods that control construction and destruction of objects.

Within a method, a function call or procedure statement allows a qualified method designator to activate a specific method. This type of call is known as a qualified method activation.

See also

[Constructors and destructors](#)

[Method activation](#)

[Method declarations](#)

[Object types](#)

[Qualified-method activation](#)

[Self](#)

Method implementations

Example

The declaration of a method within an object type corresponds to a forward declaration of that method. Somewhere after the object-type declaration, and within the same module, the method must be implemented by a defining declaration.

For procedure and function methods, the defining declaration takes the form of a normal procedure or function declaration, except that the procedure or function identifier in the heading is a qualified method identifier.

For constructors and destructors, the defining declaration takes the form of a procedure method declaration, except that the **procedure** reserved word is replaced by constructor or destructor.

A method's defining declaration can optionally repeat the formal parameter list of the method heading in the class type. The defining declaration's method heading must match exactly the order, types, and names of the parameters, and the type of the function result, if any.

In the defining declaration of a method, there is always an implicit parameter with the identifier Self, corresponding to a formal parameter of the class type. Within the method block, Self represents the instance for which the method was activated.

The scope of a component identifier in an object type extends over any procedure, function, constructor, or destructor block that implements a method of the class type.

Within a method block, the reserved word **inherited** can be used to access redeclared and overridden component identifiers. When an identifier is prefixed with **inherited**, the search for the identifier begins with the immediate ancestor of the enclosing method's object type.

Example

Given the following object type declaration,

```
type
  TFramedLabel = class(TLabel)
    protected
      procedure Paint; override;
    end;
```

the Paint method must later be implemented by a defining declaration, for example,

```
procedure TFramedLabel.Paint;
begin
  inherited Paint;
  with Canvas do
    begin
      Brush.Color := clWindowText;
      Brush.Style := bsSolid;
      FrameRect(ClientRect);
    end;
end;
```

Virtual methods

[See also](#) [Example](#)

Virtual methods are resolved by the compiler at run-time; this process is known as late binding. By default, all methods are static except constructor methods, but you can make any methods virtual by including a **virtual** directive in the method declaration.

When a virtual method is called, the actual (run-time) type of the class or object used in the method call determines which method implementation to activate.

Overriding a virtual method

An object type can override (redefine) any of the methods it inherits from its ancestors. The scope of an override method extends over all of the descendants of the defining object, or until you redefine the method identifier.

An override of a virtual method must match exactly the order, types, and names of the parameters, and the type of the function result, if any. The override must include the **override** directive in place of **virtual**.

Note: The only way to override a virtual method is through the **override** directive. If a method declaration in a descendant class specifies the same method identifier as an inherited method, but does not specify an **override** directive, the new method declaration will hide the inherited declaration, but not override it.

Example

The following two descendant classes override the Draw method inherited from TFigure.

type

```
TRectangle = class(TFigure)
  procedure Draw; override;
  :
end;
TEllipse = class(TFigure)
  procedure Draw; override;
  :
end;
```

The following section of code illustrates the effect of calling a virtual method through a class type variable whose actual type varies at run-time.

var

```
Figure: TFigure;
```

begin

```
Figure := TRectangle.Create;
Figure.Draw;                               { Invokes TRectangle.Draw }
Figure.Destroy;
Figure := TEllipse.Create;
Figure.Draw;                               { Invokes TEllipse.Draw }
Figure.Destroy;
```

end;

See also

Methods

Override directive

Virtual (standard directive)

Instantiating objects

[See also](#)

An instance of an object type is a dynamically allocated block of memory with a layout defined by the object type.

Instances of an object type are also commonly referred to as objects. Each object of an object type has a unique copy of the fields declared in the object type, but all share the same methods.

A variable of an object type contains a reference to an object of the object type. The variable does not contain the object itself, but rather is a pointer to the memory block that has been allocated for the object. Analogous to pointer variables, multiple object-type variables can refer to the same object, and an object-type variable can contain the value **nil**, indicating that it does not currently reference an object.

Note: Contrary to a pointer variable, it is not necessary to dereference an object-type variable to gain access to the referenced object. In other words, where it is necessary to write `Ptr^.Field` to access a field in a dynamically allocated record, the `^` operator is implied when accessing a component of an object, and the syntax is simply `Instance.Field`.

See also

[Constructors and destructors](#)

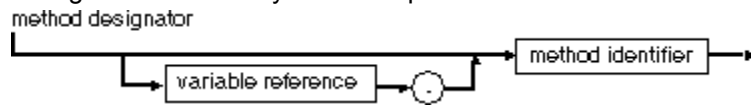
[Methods](#)

[Virtual methods](#)

Method activation

[See also](#)

You can activate a method by calling a function or procedure statement consisting of a method designator followed by an actual parameter list.



The variable reference must be an instance of an object reference or class reference, and the method identifier must be a method of that object type.

The method designator becomes an implicit actual parameter of the method. It corresponds to a formal variable parameter named Self that possesses the object type corresponding to the activated method.

When using a **with** statement, you can omit the variable-reference part of a method designator because it references the Self parameter.

Within a method declaration, you can omit the variable reference. The implicit Self parameter of the method activation becomes the Self of the method containing the call.

See also

[Function calls](#)

[Method declarations](#)

[Parameters](#)

[Procedure statements](#)

[Qualified-method activation](#)

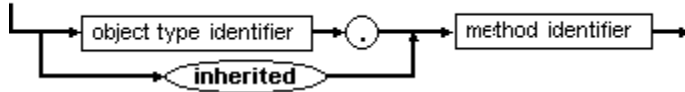
[With statement](#)

Qualified-method activations

See also [Example](#)

Qualified-method activation uses a qualifier to call a specific method. It can occur within any of the following:

- A method
 - A function call
 - A procedure statement
- qualified method designator



The object type in a qualified-method designator must be the same as the enclosing method's object type or an ancestor of it.

Use the reserved word inherited to specify the ancestor of the enclosing method's object type; you cannot use **inherited** within methods of an object type that has no ancestor.

The implicit Self parameter of a qualified-method activation becomes the Self of the method containing the call.

A qualified-method activation is always static and always invokes the specified method.

Use qualified-method activation within an override method to activate the overridden method.

See also

[Function calls](#)

[Method declarations](#)

[Object types](#)

[Procedure statements](#)

Qualified-method activation examples

The following example demonstrates a qualified-method activation that overrides a method and reuses the code of the method it overrides.

```
constructor TShape.Create(AOwner: TComponent);  
begin  
  inherited Create(AOwner);  
  Width := 65;  
  Height := 65;  
  FPen := TPen.Create;  
  FPen.OnChange := StyleChanged;  
  FBrush := TBrush.Create;  
  FBrush.OnChange := StyleChanged;  
end;
```

Component visibility

See also [Class types](#)

The visibility of a component identifier is governed by the visibility attribute of the component part in which the identifier was declared. There are five possible visibility attributes:

- published
- public
- protected
- private
- automated

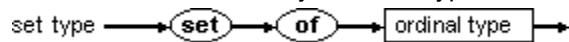
Component identifiers declared in the component list that immediately follows the object type heading have the published visibility attribute if the object type is compiled in the **{ $\$M+$ }** state or is derived from a class that was compiled in the **{ $\$M+$ }** state. Otherwise, such component identifiers have the public visibility attribute.

See also

Set types

[See also](#) [Types](#)

A set type is a collection of objects of the same ordinal type. To declare a set type use the reserved words **set of** followed by the base type.



A set type's range of values is the power set of a particular ordinal type (the base type). Each possible value of a set type is a subset of the possible values of the base type.

A variable of a set type can hold none or all the values of the set. Every set type can hold the value [], which is called the empty set.

The base type must not have more than 256 possible values, and the ordinal values of the upper and lower bounds of the base type must be within the range 0 to 255.

Comparing sets

If A and B are set operands, their comparisons produce the following results:

- A = B is True only if A and B contain exactly the same members; otherwise, A <> B.
- A <= B is True only if every member of A is also a member of B.
- A >= B is True only if every member of B is also a member of A.

Testing set membership

The relational operator **in** returns True if the value of the ordinal-type operand is a member of the set-type operand; otherwise, it returns False.

See also

[Ordinal types](#)

[Relational operators](#)

[Set \(reserved word\)](#)

[Set operators](#)

[Set-type constants](#)

[Type compatibility](#)

File types

[See also](#) [Types](#)

A file type is a linear sequence of elements that can be of any type except the following:

- File type
 - Any structured type with a file-type component
 - An object type
- The number of elements is not set by the file-type declaration.
-

If you omit the word **of** and the type from the file declaration, the file is an untyped file.

The standard file type Text (or TextFile) represents a file containing characters organized into lines.

See also

[File \(reserved word\)](#)

[Type compatibility](#)

Type compatibility

[See also](#)

Compatibility between two types is required in expressions or in relational operations. Type compatibility is a precondition of assignment compatibility.

Types are compatible when at least one of the following conditions is true:

- Both types are the same.
- Both types are real types.
- Both types are integer types.
- One type is a subrange of the other.
- Both types are subranges of the same host type.
- Both types are set types with compatible base types.
- Both types are packed string types with an identical number of components.
- One type is a string type and the other is either a string type, packed-string type, or Char type.
- One type is Pointer and the other is any pointer type.
- Both types are class types or class reference types, and one type is derived from the other.
- One type is PChar and the other is a zero-based character array of the form **array[0..X] of Char**.
- Both types are pointers to identical types. (This applies only when type-checked pointers are enabled with the `{$T+}` directive.)
- Both types are procedural types with identical result types, an identical number of parameters, and a one-to-one identity between parameter types.
- One type is Variant and the other is either an integer type, a real type, string type or Boolean type.

See also

[Assignment compatibility](#)

Assignment compatibility

[See also](#)

Assignment compatibility is necessary when a value is assigned to something, such as in an assignment statement or in passing value parameters.

An object type is assignment compatible with any ancestor object type.

A value of type T2 is assignment-compatible with a type T1 (that is, T1 := T2 is allowed) if any of the following are true:

- T1 and T2 are identical types and neither is a file type or a structured type that contains a file-type component at any level of structuring.
- T1 and T2 are compatible ordinal types, and the value of type T2 falls within the range of possible values of T1.
- T1 and T2 are real types, and the value of type T2 falls within the range of possible values of T1.
- T1 is a real type, and T2 is an integer type.
- T1 and T2 are string types.
- T1 is a string type, and T2 is a Char type.
- T1 is a string type, and T2 is a packed-string type.
- T1 is a long string type, and T2 is of type PChar.
- T1 and T2 are compatible, packed-string types.
- T1 and T2 are compatible set types, and all the members of the value of type T2 fall within the range of possible values of T1.
- T1 and T2 are compatible pointer types.
- T1 is a class type and T2 is a class type derived from T1.
- T1 is a class reference type and T2 is a class reference type derived from T1.
- T1 is a PChar and T2 is a string constant.
- T1 is a PChar and T2 is a zero-based character array of the form **array[0..X] of Char**
- T1 and T2 are compatible procedural types.
- T1 is a procedural type, and T2 is a procedure or function with an identical result type, an identical number of parameters, and a one-to-one identity between parameter types.
- T1 is Variant and T2 is an integer type, real type, string type, or Boolean type.
- T1 is an integer type, real type, string type, or Boolean type, and T2 is Variant.

A compile-time error occurs when assignment compatibility is necessary and none of the items in the preceding list are true.

See also

[Type compatibility](#)

Class reference types

See also [Example](#) [Object types](#)

Class-reference types allow operations to be performed directly on classes. This contrasts with class types, which allow operations to be performed on instances of classes. Class-reference types are sometimes referred to as metaclasses or metaclass types.

Class-reference types are useful in the following situations:

- With a virtual constructor to create an object whose actual type is unknown at compile time
- With a class method to perform an operation on a class whose actual type is unknown at compile time
- As the right operand of an is operator to perform a dynamic type check with a type that is unknown at compile time
- As the right operand of an as operator to perform a checked typecast to a type that is unknown at compile time

The declaration of a class-reference type consists of the reserved words **class of** followed by a class-type identifier. For example,

```
type
  TComponent = class(TPersistent)
  :
  end;
  TComponentClass = class of TComponent;
  TControl = class(TComponent)
  :
  end;
  TControlClass = class of TControl;
var
  ComponentClass: TComponentClass;
  ControlClass: TControlClass;
```

The previous declarations define TComponentClass as a type that can reference class TComponent, or any class that derives from TComponent, and TControlClass as a type that can reference class TControl, or any class that derives from TControl.

Class-type identifiers function as values of their corresponding class-reference types. For example, in addition to its other uses, the TComponent identifier functions as a value of type TComponentClass, and the TControl identifier functions as a value of type TControlClass.

A class-reference type value is assignment-compatible with any ancestor class-reference type. Therefore, during program execution, a class-reference type variable can reference the class it was defined for or any descendant class of the class it was defined for. Referring to the previous declarations, the assignments

```
ComponentClass := TComponent;      { Valid }
ComponentClass := TControl;        { Valid }
```

are both valid. Of these assignments,

```
ControlClass := TComponent;        { Invalid }
ControlClass := TControl;          { Valid }
```

only the second one is valid, however. The first assignment is an error because TComponent is not a descendant of TControl, and therefore not a value of type TControlClass.

A class-reference type variable can be **nil**, which indicates that the variable does not currently reference a class.

Every class inherits (from TObject) a method function called ClassType, which returns a reference to the class of an object. The type of the value returned by ClassType is TClass, which is declared as **class of** TObject. This means that the value returned by ClassType may have to be typecast to a more specific descendant type before it can be used, for example

```
if Control <> nil then  
  ControlClass := TControlClass(Control.ClassType) else  
  ControlClass := nil;
```

See also

[Class \(reserved word\)](#)

Example

type

```
{A variable of type TComponentRef can be set at run time to refer  
to TComponentClass or any of its subclasses}  
TComponentRef = class of TComponentClass;
```

var

```
TRef: TComponentRef;  
NewComponent: TComponent;
```

...

```
TRef := TButton; {TRef can now be used anywhere the name of the  
class TButton could be used}  
NewComponent := TRef.Create; {TRef is used here to create a new  
TButton}
```

...

Null-terminated strings

[See also](#)

C++Builder supports a class of character strings called null-terminated strings. Null-terminated strings are widely used by the C and C++ programming languages, and by Windows itself. Using C++Builder's null-terminated string support and the null-terminated string handling functions supplied by the SysUtils unit, you can easily interface with other languages and the Windows API.

What is a null-terminated string?

A null-terminated string consists of a sequence of non-null characters followed by a NULL (#0) character. Unlike Object Pascal strings, null-terminated strings have no separate length indicator. Instead, the first NULL character in a null-terminated string marks the end of the string.

Using null-terminated strings

Null-terminated strings are stored as arrays of characters with a zero-based integer index type, like `array[0..X] of Char`

where X_ = a positive nonzero integer.

This is called a zero-based character array.

Null-terminated strings use character pointers with

- [String literals](#)
- [Character arrays](#)

Null-terminated strings and standard procedures

The following standard procedures can be applied to zero-based character arrays:

- Read
- Readln
- Str
- Val

The following procedures can also be applied to character pointers:

- AssignFile
- Rename
- Val
- Write
- Writeln

See also

[String types](#)

[Character pointer indexing](#)

[Null-terminated string functions](#)

[Mixing long strings and null-terminated strings](#)

[Null-terminated wide character strings](#)

Character pointers and string literals

[See also](#) [Example](#) [Null-terminated strings](#)

A string literal is assignment-compatible with the PChar type. This means that a string literal can be assigned to a variable of type PChar.

The effect of such an assignment is that the pointer points to an area of memory that contains a null-terminated copy of the string literal.

You can use string literals as actual parameters in procedure and function calls when the corresponding formal parameter is of type PChar. Just as it does with an assignment, the compiler generates a null-terminated copy of the string literal.

You can initialize a typed constant of type PChar with a string constant. You can do this with structured types as well, such as arrays of PChar and records and objects with PChar fields.

See also

[Character pointers and character arrays](#)

[Character pointer indexing](#)

[Null-terminated strings](#)

Example**var**

P: PChar;

begin

P := 'Hello world...';

end;

Character pointers and character arrays

[See also](#) [Example](#) [Null-terminated strings](#)

A zero-based character array is compatible with the PChar type. This means that whenever a PChar is expected, you can use a zero-based character array instead.

When you use a character array in place of a PChar value, the compiler converts the character array to a pointer constant whose value corresponds to the address of the first element of the array.

You can initialize a typed constant of a zero-based character array type with a string literal that is shorter than the declared length of the array. The remaining characters are set to NULL (#0), and the array effectively contains a null-terminated string.

See also

[Character pointers and character arrays](#)

[Character pointer indexing](#)

[Null-terminated strings](#)

Example

var

```
A: array[0..63] of Char;  
P: PChar;
```

begin

```
P := A;           {P now points to the first element of A}  
PrintStr(A);  
PrintStr(P);     {PrintStr is called twice with the same value}
```

end;

Character pointer indexing

[See also](#) [Example](#) [Null-terminated strings](#)

Since a zero-based character array is compatible with a character pointer, a character pointer can be indexed as if it were a zero-based character array.

When you index a character pointer, the index specifies a signed offset to add to the pointer before it is dereferenced. Therefore, $P[0] = P^{\wedge}$. $P[0]$ specifies the character pointed to by P ; $P[1]$ specifies the character right after the one pointed to by P ; $P[2]$ specifies the next character, and so on. Likewise, $P[-1]$ specifies the character right before the one pointed to by P , and so on.

The compiler performs no range checks when indexing a character pointer because it has no type information available to determine the maximum length of the null-terminated string pointed to by the character pointer. Your program must perform any such range checking.

Example

{The following example uses character pointer indexing to convert a null-terminated string to uppercase.}

```
function StrUpper(Str: PChar): PChar;  
var  
    I: Integer;  
begin  
    I := 0;  
    while Str[I] <> #0 do  
        begin  
            Str[I] := UpCase(Str[I]);  
            Inc(I);  
        end;  
    StrUpper := Str;  
end;
```

See also

[PChar](#)

Null-terminated string functions

[Null-terminated strings](#)

The SysUtils unit provides a number of null-terminated string handling functions. The following table gives a brief description of each of these functions.

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using StrAlloc or StrNew.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using StrAlloc or StrNew.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lower case.
StrMove	Moves a block of characters from one string to another.
StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to upper case.

Mixing long strings and null-terminated strings

See also [Null-terminated strings](#)

Object Pascal allows you to mix long strings and null-terminated strings in expressions and assignments. The rules that control mixing of long strings and null-terminated strings are outlined below.

- A null-terminated string is assignment compatible with a long string. In other words, a PChar value can be assigned to a variable of type **string**, or passed as a constant or value parameter of type **string**. For example, 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 an expression, if one operand of a binary operator is of type **string** and the other is of type PChar, the PChar operand is automatically converted to type **string**.
- A typecast of the form **string**(P), where P is a PChar expression, can be used to explicitly convert a null-terminated string to a long string. This is useful in situations where both operands of an operator are of type PChar, but you want to perform a string operation. For example,

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

concatenates two null-terminated strings to form a resulting long string.
- A typecast of the form PChar(S), where S is a long string expression, can be used to convert a long string to a null-terminated string.

See also

[Long string types](#)

Null-terminated wide character strings

See also [Null-terminated strings](#)

The Windows operating system supports three types of character sets:

- single-byte character sets
- double-byte character sets
- the Unicode character set

Single-byte characters

With a single-byte character set (SBCS), a character string is a sequence of bytes, and each byte represents an individual character.

The ANSI character set used by most Western versions of Windows is a single-byte character set.

Double-byte characters

With a double-byte character set (DBCS), a character string is also a sequence of bytes, but unlike a single-byte character set, some characters are represented by one byte and others by two bytes.

The first byte of a two byte character is called the lead byte.

In general, the lower 128 characters of a double-byte character set map to the 7-bit ASCII character set, and lead bytes typically have ordinal values greater than or equal to 128.

Double-byte character sets are widely used in Asia, where native character sets contain far more than 256 characters.

Unicode characters

The Unicode character set is fundamentally different from single- and double-byte character sets in that each character is represented as a word (two bytes). A character string in the Unicode character set is a sequence of words, not bytes.

Unicode characters are also known as wide characters, and Unicode strings are often referred to as wide character strings.

With 65536 possible values for each character, Unicode can represent all the world's characters in modern computer use, including technical symbols and special characters used in publishing.

The first 256 characters of the Unicode character set map to the ANSI character set.

C++Builder character sets

C++Builder supports single- and double-byte characters and character strings through the `AnsiChar`, `PAnsiChar`, and `AnsiString` fundamental types, and the `Char`, `PChar`, and `string` generic types. Wide characters are supported through the `WideChar` and `PWideChar` types.

There is no wide character equivalent of the `string` type.

Strings of wide characters

The null-terminated string handling features of Object Pascal also apply to the `PWideChar` type. This means that a string literal is assignment compatible with the `PWideChar` type (although the string literal can only contain wide characters with ordinal values in the ANSI character range). Also, a zero-based wide character array of the form

```
array[0..X] of WideChar
```

is assignment compatible with the `PWideChar` type, and a value of type `PWideChar` can be indexed as if it were a zero-based wide character array. Furthermore, the [character-pointer operators](#) (+ and -) also apply to wide character pointers.

Note When adding or subtracting integer offsets to and from wide character pointers, the offsets represent distances in wide characters, and are therefore automatically multiplied by two before being added to or subtracted from the pointers. Likewise, when subtracting one wide character pointer from another, the resulting integer is automatically divided by two to yield a distance in wide characters.

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.

See also

[Character types](#)

[String types](#)

[Null-terminated string functions](#)

Variant types

[See also](#)

[Examples](#)

[Types](#)

The Variant type is capable of representing values that change type dynamically. Whereas a variable of any other type is statically bound to that type, a variable of the Variant type can assume values of differing types at run-time. The Variant type is most commonly used in situations where the actual type to be operated upon varies or is unknown at compile-time.

The predefined identifier Variant is used to denote the variant type. Variants have the following characteristics:

- Variants can contain integer values, real values, string values, boolean values, date-and-time values, and OLE Automation objects. In addition, variants can contain arrays of varying size and dimension with elements of any of these types.
- The special variant value Unassigned is used to indicate that a variant has not yet been assigned a value, and the special variant value Null is used to indicate unknown or missing data.
- Variants can be combined with other variants and with integer, real, string, and boolean values in expressions, and the compiler will automatically generate code that performs the necessary type conversions.
- When a variant contains an OLE Automation object, the variant can be used to get and set properties of the object, and to invoke methods on the object.
- Variant variables are always initialized to be Unassigned when they are first created. This is true whether a variant variable is global, local, or part of a structure such as an array, record, or object.

Note that while variants offer great flexibility, they also consume more memory than regular variables, and operations on variants are substantially slower than operations on statically typed values.

Variant type examples

The following section of code demonstrates the use of variants and some of the automatic type conversions that are performed when variants are combined 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 }
  D := V2;         { D = 1234.5678 }
  S := V3;         { S = 'Hello world' }
  I := V4;         { I = 1000 }
  S := V5;         { S = '2235.5678' }
end;
```

See also

[Values in variants](#)

[Variant type conversions](#)

[Variant expressions](#)

[Variant arrays](#)

[Variants and OLE Automation objects](#)

Values in variants

Variant types

A variable of type Variant occupies 16 bytes of memory, and its internal representation consists of a type code and a value (or a reference to a value) of the type given by the type code. The standard function VarType returns a variant's type code. The following table lists the variant type constants and values, and the meaning of each type code.

VarType	Value	Contents of variant
varEmpty	\$0000	The variant is Unassigned. The variant has not been assigned a value and is assumed to be zero. When used in expressions, VarEmpty is coerced to a zero value or empty string.
varNull	\$0001	The variant is Null. The variant has not been assigned a value. VarNulls propagate through expressions. An expression containing a VarNull variant and an assigned variant will evaluate to VarNull. VarNull is usually used to indicate missing data, a state which should propagate through all calculations on that data.
varSmallint	\$0002	16-bit signed integer (type Smallint).
varInteger	\$0003	32-bit signed integer (type Integer).
varSingle	\$0004	Single-precision floating-point value (type Single).
varDouble	\$0005	Double-precision floating-point value (type Double).
varCurrency	\$0006	Currency floating-point value (type Currency). The variant contains a currency value (an 8 byte fixed point value with 4 decimal places)
varDate	\$0007	Date and time value (type TDateTime).
varOleStr	\$0008	Reference to an OLE string (a dynamically allocated Unicode string). The variant contains a pointer to a null terminated string allocated with SysAllocStr.
varDispatch	\$0009	Reference to an OLE automation object (an IDispatch interface pointer).
varError	\$000A	Operating system error code.
varBoolean	\$000B	16-bit boolean (type WordBool).
varVariant	\$000C	Variant (used only with variant arrays).
varUnknown	\$000D	Reference to an unknown OLE object (an IUnknown interface pointer). The variant contains indeterminate or custom data.
varByte	\$0011	8-bit unsigned integer (type Byte).
varString	\$0100	Reference to a dynamically-allocated long string (type AnsiString).
varTypeMask	\$0FFF	Bit mask for extracting type code. This constant is a mask that can be combined with the VType field using a bit-wise AND..
varArray	\$2000	Bit indicating variant array. This constant is a mask that can be combined with the VType field using a bit-wise AND to determine if the variant contains a single value or an array of values.
varByRef	\$4000	This constant can be AND'd with Variant.VType to determine if the variant contains a pointer to the indicated data instead of containing the data itself.

The varXXXX constants returned by the VarType standard function are defined in the System unit. Note that future versions of C++Builder may define additional type codes, so be careful not to write code that

depends on only these codes being returned.

The `varArray` bit position is set if the variant contains an array of the given type. The `varTypeMask` bit mask is used to extract the actual type code from a value returned by the `VarType` function. For example, the following expression is True if V contains a Double or an array of Double:

```
if VarType (V) and varTypeMask = varDouble then ...
```

The `TVarData` record defined in the System unit can be used to typecast a Variant variable to gain access to its internal representation. For further details, see the description of `TVarData` in the Visual Component Library Reference manual.

Variant type conversions

See also [Variant types](#)

All integer, real, string, character, and boolean types are assignment compatible with the Variant type. The following table lists the types that can be assigned to a Variant, and the resulting variant type codes.

Expression type	Variant type code
integer types	varInteger
real types except Currency	varDouble
Currency type	varCurrency
string and character types	varString
Boolean types	varBoolean

An expression can be explicitly cast to type Variant using a typecast of the form Variant(X), where X is an expression of one of the types listed in the table above.

A Variant is assignment compatible with all integer, real, string, and Boolean types. The following tables show the type conversion rules that govern conversions of Variant values to other types.

Converting a variant to an integer type value

Variant type	Result
varEmpty	0.
varNull	Raises an EVariantError exception.
varByte	Converts one integer format to another, raises an EVariantError exception if value does not fit in destination format.
varSmallint	
varInteger	
varError	
varSingle	Rounds real value to nearest integer, raises an EVariantError exception if result does not fit in destination format.
varDouble	
varCurrency	
varDate	Interprets date and time value as a Double, rounds value to nearest integer, raises an EVariantError exception if result does not fit in destination format.
varOleStr	Converts string to integer, raises an EVariantError exception if string is not a valid integer value or if result does not fit in destination format.
varString	
varBoolean	0 for False, -1 for True (255 if destination type is Byte).

Converting a variant to a real type value

Variant type	Result
varEmpty	0.
varNull	Raises an EVariantError exception.
varByte	Converts integer to real.
varSmallint	
varInteger	

varError	
varSingle	Converts from one real format to another, raises an EVariantError exception if value does not fit in destination format.
varDouble	
varCurrency	
varDate	Interprets value as a Double, converts value to destination format, raises an EVariantError exception if value does not fit in destination format.
varOleStr	Converts string to real using the Regional Settings in the Windows Control Panel, raises an EVariantError exception if string is not a valid real value or if result does not fit in destination format.
varString	
varBoolean	0 for False, -1 for True.

Converting a variant to a date and time value

Variant type	Result
varEmpty	12/30/1899 12:00:00 am.
varNull	Raises exception.
varByte	Converts integer to Double, interprets result as a date and time value.
varSmallint	
varInteger	
varError	
varSingle	Converts value to Double, interprets result as a date and time value.
varDouble	
varCurrency	
varDate	No translation.
varOleStr	Converts string to a date and time value using the Regional Settings in the Windows Control Panel.
varString	
varBoolean	Converts Boolean to Double, interprets result as a date and time value.

Converting a variant to a string type value

Variant type	Result
varEmpty	Empty string.
varNull	Raises exception.
varByte	Converts integer value to its string representation.
varSmallint	
varInteger	
varError	
varSingle	Converts real value to its string representation using the Regional Settings in the Windows Control Panel.
varDouble	
varCurrency	
varDate	Converts date and time value to its string representation using the Regional Settings in the Windows Control Panel.
varOleStr	Converts from Unicode to ANSI if destination type is varString.

varString	Converts from ANSI to Unicode if destination type is varOleStr.
varBoolean	'0' for True, '-1' for False.

Converting a variant to a Boolean type value

Variant type	Result
varEmpty	False.
varNull	Raises exception.
varByte	False if value is zero, True if value is non-zero.
varSmallint	
varInteger	
varError	
varSingle	False if value is zero, True if value is non-zero.
varDouble	
varCurrency	
varDate	Interprets value as a Double, returns False if value is zero, True if value is non-zero.
varOleStr	False if string contains 'false' (case insensitive) or a numeric string that evaluates to zero, True if string contains 'true' (case insensitive) or a numeric string that evaluates to non-zero, otherwise raises an EVariantError exception.
varString	
varBoolean	No translation.

A Variant value can be explicitly cast to an integer, real, string, or Boolean type using a typecast of the form TypeName(V), where TypeName is an integer, real, string, or Boolean type identifier and V is an expression of type Variant. Furthermore, the VarAsType standard function and the VarCast standard procedure can be used to change the internal representation of a variant. The tables above list the rules that govern all such type conversions.

If a variant contains a reference to an OLE Automation object (the varDispatch type code), any attempt to convert the variant to another type will first fetch the value of the object's default property and then convert that value to the requested type. If the given OLE Automation object has no default property, an EVariantError exception is raised.

See also

[Values in variants](#)

[Variant expressions](#)

Variant expressions

See also [Variant types](#)

Variants can participate in expressions. The following operators support operands of type Variant:

`+` `-` `*` `/` `div` `mod` `shl` `shr` `and` `or` `xor` `not` `=` `<>` `<` `>` `<=` `>=`

For operators that take two operands, if one operand is of type Variant, the other operand is automatically converted to type Variant using the rules set forth in [Variant type conversions](#). The result type of a non-relational operation (all operators shown above but the last six) on Variant values is always Variant. The result type of a relational operation on Variant values is always Boolean.

For all non-relational operators, if one or both operands are Unassigned, an EVariantError exception is raised. In other words, no operations other than comparisons are allowed on Unassigned variants.

Also for all non-relational operators, if one or both operands are Null, the result of the operation is Null. In other words, Null values propagate through expressions, and the presence of a Null value in an expression causes the entire expression to become Null.

When performing a double operand operation, the common type of the two Variant operands governs the operation. The common type is determined using the matrix shown in the table below. When reading this table, variant type codes varSmallint, varInteger, and varByte map to Integer, varSingle and varDouble map to Double, and varOleStr and varString map to String.

Variant operation type matrix

	Integer	Double	Currency	String	Boolean	Date
Integer	Integer	Double	Currency	Double	Integer	Date
Double	Double	Double	Currency	Double	Double	Date
Currency	Currency	Currency	Currency	Currency	Currency	Date
String	Double	Double	Currency	String	Boolean	Date
Boolean	Integer	Double	Currency	Boolean	Boolean	Date
Date	Date	Date	Date	Date	Date	Date

For example, in the operation `V1 + V2`, if the type code of `V1` is `varInteger` and the type code of `V2` is `varString`, the common type used to perform the operation is `Double`.

For non-relational operators, once the common type is established the operation proceeds as described in the following table.

Non-relational variant operator results

Common type	Operator results
Integer	For all operators except <code>/</code> , the operands are converted to Integer, and the result type is Integer, or Double if the result does not fit in a 32-bit signed integer. For the <code>/</code> operator, the operation is performed as a Double operation.
Double	For the <code>+</code> , <code>-</code> , <code>*</code> , and <code>/</code> operators, the operands are converted to Double and the result type is Double. For all other operators, the operation is performed as an Integer operation.
Currency	For the <code>+</code> , <code>-</code> , <code>*</code> , and <code>/</code> operators, the operands are converted to Currency and the result type is Currency, or Double when dividing two Currency values. For all other operators, the operation is performed as an Integer operation.
String	For the <code>+</code> operator, if both operands are strings, the result is the

concatenation of the two strings. Otherwise, and for all other operators, the operation is performed as a Double operation.

Boolean For the and, or, and xor operators, the operands are converted to Boolean and the result type is Boolean. For all other operators, the operation is performed as a Double operation.

Date For the + and – operators, the operands are converted to Date and the result type is Date, or Double when subtracting two Date values. For all other operators, the operation is performed as a Double operation.

For relational operators, both variants are converted to the common type, and the resulting values are compared to produce a Boolean result. The Unassigned value compares less than all other values. The Null value compares greater than Unassigned and less than all other values.

For the unary minus (–) operator, strings are converted to Double before the operation, and booleans are converted to Integer before the operation.

For the **not** operator, if the type code of the variant is varBoolean, a logical negation operation is performed. For all other type codes, the variant is converted to Integer and a bitwise negation operation is performed.

See also

[Variant type conversions](#)

[Values in variants](#)

Variant arrays

[See also](#) [Example](#) [Variant types](#)

Variants can contain arrays of varying size and dimension with elements of any of the variant base types. The elements of a variant array are all of the same type, but if the element type is Variant, individual elements can of course contain different kinds of data (including other variant arrays).

Variant arrays are typically created using the VarArrayCreate standard procedure.

The following table lists the variant array standard procedures and functions which are all defined in the System unit.

Variant array standard procedures and functions

Name	Description
VarArrayCreate	Creates a variant array with a given low and high bound for each dimension and a given element type. The element type can be any of the varXXXX type codes, except varString. To create a variant array of strings, you must use the varOleStr type code. The elements of the newly created array are all set to zero or empty.
VarArrayOf	Creates a one-dimensional variant array with a given list of elements. The element type of the returned variant array is Variant. The VarArrayOf function is useful for "on the fly" construction of variant array parameters.
VarArrayRedim	Resizes a variant array by changing the high bound of the rightmost dimension to a given value. Existing elements of the array are preserved, and new elements are set to zero or empty.
VarArrayDimCount	Returns the number of dimensions in a variant array, or zero if the argument is not a variant array.
VarArrayLowBound	Returns the low bound of a given dimension in a variant array.
VarArrayHighBound	Returns the high bound of a given dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data in the variant array. Using this function you can gain direct access to the data in a variant array for much improved performance.
VarArrayUnlock	Unlocks a variant array that was previously locked by VarArrayLock.
VarIsArray	Tests whether the argument contains a variant array.

Note The element type of a variant array cannot be varString. To create variant arrays of strings, you must use the varOleStr type code.

When a variant contains an array, elements of the array can be accessed by following the variant with one or more index expressions, enclosed in square brackets and separated by commas. Index expressions are always of type Integer. When indexing a variant, an EVariantError exception is raised if the variant does not contain a variant array, if an incorrect number of index expressions are specified, or if one or more of the index expressions are not within the bounds of the corresponding dimension.

Variant array elements can be accessed in expressions and assigned new values using assignment statements. Note however, that it is not possible to pass a variant array element as a **var** parameter.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, a copy of the entire array is made, possibly consuming a lot of memory. For this reason, whenever possible you should avoid assigning variant arrays to other variants, and unless there is a good reason not to, you should pass variant arrays as **var** or **const** parameters.

Variant array example

Variant arrays are typically created using the VarArrayCreate standard procedure, for example:

```
var
  A, B: Variant;
  I: Integer;
begin
  A := VarArrayCreate([0, 9], varInteger);
  for I := 0 to 9 do A[I] := I * I;
  B := VarArrayCreate([1, 3, 0, 9], varVariant);
  for I := 0 to 9 do B[1, I] := I;
  for I := 0 to 9 do B[2, I] := Sqrt(I);
  for I := 0 to 9 do B[3, I] := Format('Value=%d', [I]);
  ...
end;
```

See also

[Variant array dimensions](#)

[Locking variant arrays](#)

Variant array dimensions

[See also](#) [Example](#) [Variant arrays](#)

A variant array can be resized using the `VarArrayRedim` standard procedure. `VarArrayRedim` allows you to change the high bound of the rightmost (last) dimension of a variant array. The bounds of other dimensions cannot be changed. Existing elements of an array are preserved across a resize operation.

The `VarArrayDimCount`, `VarArrayLowBound`, and `VarArrayHighBound` standard functions allow you to examine the number of dimensions in a variant array, and the bounds of each dimension. This may be useful when writing general purpose variant array manipulation routines, such as the `VarArraySum` function shown on the example screen.

Variant array dimension examples

The code fragment below demonstrates the use of `VarArrayRedim`.

```
var
  A: Variant;
  I: Integer;
begin
  A := VarArrayCreate([0, 4], varOleStr);
  for I := 0 to 4 do A[I] := 'Initial';
  ...
  VarArrayRedim(A, 9);
  for I := 5 to 9 do A[I] := 'Additional';
  ...
end;

function VarArraySum(const A: Variant): Double;
var
  I: Integer;
begin
  if VarArrayDimCount(A) <> 1 then
    raise Exception.Create('One-dimensional variant array expected');
  Result := 0;
  for I := VarArrayLowBound(A, 1) to VarArrayHighBound(A, 1) do
    Result := Result + Double(A[I]);
end;
```

See also

[Locking variant arrays](#)

Locking variant arrays

[See also](#) [LockingVariantArraysSA](#) [Example](#) [LockingVariantArraysEx](#) [Variant arrays](#) [VariantArrays](#)

Using the `VarArrayLock` standard function and the `VarArrayUnlock` standard procedure you can gain direct access to the data in a variant array.

Variant arrays with an element type of `varByte` (like the one created by the function above) are the preferred method of passing binary data between OLE 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.

Variant-array locking example

The VarArrayLoadFile function shown below loads the contents of a file into a variant array of bytes. It uses VarArrayLock and VarArrayUnlock to read the file directly into the array.

```
function VarArrayLoadFile(const FileName: string): Variant;  
var  
    F: file;  
    Size: Integer;  
    Data: PChar;  
begin  
    AssignFile(F, FileName);  
    Reset(F, 1);  
    try  
        Size := FileSize(F);  
        Result := VarArrayCreate([0, Size - 1], varByte);  
        Data := VarArrayLock(Result);  
        try  
            BlockRead(F, Data^, Size);  
        finally  
            VarArrayUnlock(Result);  
        end;  
    finally  
        CloseFile(F);  
    end;  
end;
```

See also

[Variant array dimensions](#)

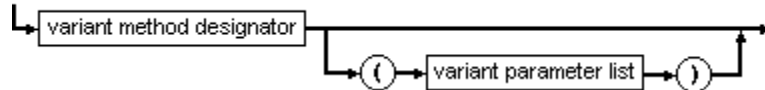
Variants and OLE Automation objects

[See also](#) [Example](#) [Variant types](#)

When a variant contains a reference to an OLE Automation object, C++Builder allows you to call methods and get and set properties of the object. You enable this functionality by using the OleAuto unit, that is by including a reference to OleAuto in the **uses** clause of one of your units or in the **uses** clause of your program or library.

The syntax of an OLE Automation object method call or property access is analogous to that of a normal method call or property access.

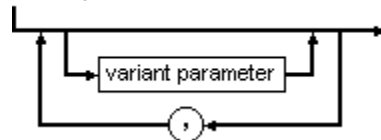
variant method call



variant method designator



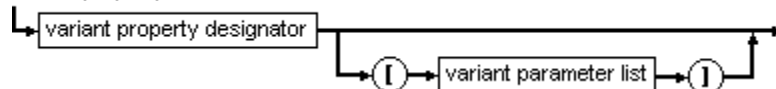
variant parameter list



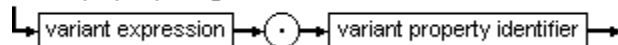
variant parameter



variant property access



variant property designator



An OLE Automation object method call is late bound, and requires no previous declaration of the method and its parameters. Contrary to a normal method call, in an OLE Automation method call the compiler allows any method identifier and any number and type of parameters to be specified. Whether or not the method call will actually succeed is not determined until it is executed at run-time.

Identifiers for OLE Automation object methods, properties, and named parameters are allowed to contain alphabetical characters from an international character set such as á, ü, and Ø.

OLE automation variant example

An example of a section of code that uses OLE Automation method calls follows below. Notice the use of the CreateOleObject function (defined in the OleAuto unit) to create a variant that contains a reference to an OLE Automation object.

```
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;
```

See also

[Variant OLE automation parameters](#)

[OLE automation properties](#)

Variant OLE automation parameters

[See also](#)

[Example](#)

[Variants and OLE Automation](#)

OLE Automation method calls support two types of parameters:

- positional parameters
- named parameters.

A positional parameter is simply an expression. A named parameter consists of a parameter identifier followed by a colon-equals symbol (:=) followed by an expression.

Positional parameters must precede named parameters in an OLE Automation method call. It is possible to omit positional parameters by leaving parameter positions empty and by writing fewer parameter expressions than are expected by the method call.

Note OLE Automation servers often do not support named parameters. Likewise, some OLE Automation servers do not allow you to omit positional parameters.

OLE Automation method call parameters can be of the following types: Integer, real, string, Boolean, and variant. 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 (or **string**), WordBool, or Variant. If a parameter expression is not just a variable reference, or if the expression is not of one of the above types, the parameter is passed by value. When a parameter is passed by reference, the corresponding variable can be modified by the method to which a call is made.

Note Passing a parameter by reference to a method that expects a value parameter simply causes OLE to fetch the value from reference parameter. The reverse situation however causes an error. In other words, it is an error to pass a value parameter where a reference parameter is expected.

OLE automation method parameters examples

Some examples of positional and named parameters follow below.

```
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');
```

The first call supplies one positional parameter. The second call supplies two positional parameters. The third call supplies four positional parameters, whereof the middle two are omitted. The fourth call supplies one positional parameter and one named parameter. Finally, the fifth call supplies no positional parameters and two named parameters. Note that using named parameters it is possible to write the parameters in any order.

See also

[OLE automation properties](#)

OLE automation properties

See also [Variants and OLE automation](#)

Access to properties of an OLE Automation object follows the same rules as method calls. When a property access is used in an expression, the value of the property is read, and when a property access is used on the left hand side in an assignment statement, the value of the property is written. For array properties, the index parameters list must be enclosed in square brackets.

The C++Builder compiler allows any method or property identifier and any number and type of parameters to be specified in an OLE Automation method call or property access. The call information is packaged up by the compiler, and not until the call or property access is executed at run-time is it known whether it will succeed.

If an OLE Automation method call or property access fails, an `EOleError` exception is raised. A call or property access may fail for any of the reasons listed below.

- The variant expression specified in the variant method designator or variant property designator does not currently reference an OLE Automation object.
- The method or property identifier is not supported by the OLE Automation object.
- An incorrect number of parameters were specified, or the type(s) of one or more parameters were incorrect.
- One or more required positional parameters were omitted.
- Named parameters were specified, but are not supported by the OLE Automation object.
- A method call was used in an expression, but the method did not return a value.
- The method or property access was successfully called, but returned an exception.

When an OLE Automation method call or property access fails, the `EOleError` exception object contains an error message that explains the reason for the failure.

See also

[Variant OLE automation parameters](#)

