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

About the built-in assembler

With Borland's built-in assembler, you can write Intel 80x86 assembler code directly inside your Pascal programs.

Index to built-in assembler Help

Using the built-in assembler

Assembler statements access the built-in assembler using the asm directive.

```
asm
  AssemblerStmt <Separator AssemblerStmt>
end
```

Built-in assembler procedures and functions must obey the same rules as external procedures and functions.

Expressions

Borland's built-in assembler operands are expressions. The basic elements of an expression are constants, registers, symbols, and operators.

The built-in assembler divides expressions into three classes:

Registers

Memory references

Immediate values

Symbols

The built-in assembler provides access to Pascal symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions.

In addition to any currently declared Pascal types, the built-in assembler provides several predefined type symbols.

Constants

Borland's built-in assembler supports two types of constants:

Numeric constants

String constants

Opcodes, operators, and directives

Borland's built-in assembler supports:

- All 8086/8087, 80286/80287, 80386/80387 instructions
- Opcodes
- Most Turbo Assembler expression operators

- Turbo Assembler's define byte, define word, and define double word directives (DB, DW, and DD)

Most operations implemented through Turbo Assembler directives (such as EQU, PROC, STRUC, and MACRO) are matched by corresponding Object Pascal constructs. For example, most EQU directives correspond to **const**, **var**, and **type** declarations, and STRUC corresponds to **record** types.

The built-in assembler also implements a large subset of the syntax supported by the Borland Turbo Assembler and the Microsoft Macro Assembler.

Built-in assembler entry and exit code

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
```

where

- Locals is the size of the local variables.
- Params is the size of the parameters.

If both Locals and Params are 0, there is no entry code, and the exit code consists simply of a RET instruction.

Built-in assembler operands

Built-in assembler operands are expressions made up of constants, registers, symbols, and operators.

Although built-in assembler expressions are built using the same basic principles as Object Pascal expressions, there are some important differences.

The built-in assembler

- Recognizes its own set of reserved words.
- Evaluates all expressions as 32-bit integer values.
- Interprets variable references as the address of the variable.

Note: Pascal interprets variable references as the variable's contents.

Expressions must also resolve to a constant value.

Built-in assembler labels

Example

Labels defined in assembler are identifiers. Only the first 32 characters of a label are significant in the built-in assembler.

Nonlocal labels you define in assembler language must be declared in a label declaration part of the block containing the **asm** statement.

Local labels

A local label is not declared in the label declaration part but is known only within its declaring **asm** statement.

Local labels start with an at-sign (@). Since the at-sign cannot be part of an Object Pascal identifier, local labels are automatically restricted to use within **asm** statements. The scope of a local label is within its defining **asm** statement.

The exact syntax for a local label identifier is an at-sign (@) followed by any combination of one or more of the following symbols:

- Letters (A..Z)
- Digits (0..9)
- Underscores (_)
- At-signs (@)

The end of the identifier is denoted by a colon (:).

Example

The following example declares the label `BigJump`. Notice that there are no local labels declared in this example.

```
procedure foo;
label
  BigJump;
begin
  asm
    mov ax, SomeVar
    or  ax, ax
    JZ  BigJump
    mov SomeVar, dx
  end
  writeln(SomeVar);
  asm
BigJump:
    mov    ax, dx
  end;
end;
```

The following example declares local labels.

```
procedure foo;
begin
  asm
    or  ax, ax
    JZ  @@MyLabel
    mov dx, ax
    mov cx, ax
  @@MyLabel:
    inc ax
  end;
end;
```

Expression classes

The built-in assembler divides expressions into three classes:

Register expressions

Memory reference expressions

Immediate value expressions

Built-in assembler expressions

The built-in assembler evaluates all expressions as 32-bit integer values. It does not support floating-point and string values, except string constants.

Built-in assembler expressions are built from expression elements (constants, registers, and symbols) and operators, and each expression has an associated expression class and expression type.

Differences between Pascal and assembler expressions

There are two major differences between an Object Pascal expression and an Assembler expression. They are:

- All built-in assembler expressions must resolve to a constant value, a value that can be computed at compile time.
- Variable references in a built-in assembler expression denote the address of the variable. For example, in Object Pascal, the expression X + 4, where X is a variable, means the contents of X plus 4, whereas in the built-in assembler, it means the contents of the word at an address four bytes higher than the address of X.

Built-in assembler expression operators

This table lists the built-in assembler's expression operators in decreasing order of precedence.
The operators within each category have equal precedence.

Category	Operator	What it is (or does)
Highest	&	Identifier override
	(...)	Subexpression
	[...]	Memory reference
	.	Structure member selector
Unary	HIGH	Returns high-order 8 bits
	LOW	Returns low-order 8 bits
	+	Unary plus
	-	Unary minus
	:	Segment override
	OFFSET	Returns offset part
	TYPE	Returns type (byte size)
	PTR	Typecast
	*	Multiplication
	/	Integer division
	MOD	Integer modulus (remainder)
	SHL	Logical shift left
	SHR	Logical shift right
Additive	+	Binary addition
	-	Binary subtraction
Bitwise	NOT	Bitwise negation
	AND	Bitwise AND
	OR	Bitwise OR
	XOR	Bitwise Exclusive OR

Built-in assembler expression types

Every built-in assembler expression has an associated type.

This type is a size, because the built-in assembler regards the type of an expression as the size of its memory location.

The built-in assembler performs type checking whenever possible; an error results if the type check fails.

You can use a typecast to change the type of a memory reference. For example, all of these refer to the first (least significant) byte of the OutBufPtr variable:

```
asm
  MOV     DL,BYTE PTR OutBufPtr
  MOV     DL,Byte(OutBufPtr)
  MOV     DL,OutBufPtr.Byte
end;
```

In some cases, a memory reference is untyped (it has no associated type), for example, an immediate value enclosed in brackets:

```
asm
  MOV     AL,[100H]
  MOV     BX,[100H]
end;
```

The built-in assembler permits both of these instructions because the expression `[100H]` has no associated type (it just means "the contents of address 100H in the data segment"), and the type can be determined from the first operand (BYTE for AL, WORD for BX).

In cases where the type cannot be determined from another operand, the built-in assembler requires an explicit typecast, as in the following example:

```
asm
  INC     BYTE PTR [100H]
  IMUL    WORD PTR [100H]
end;
```

Built-in assembler predefined type symbols

In addition to any currently declared Pascal types, the built-in assembler provides these predefined type symbols.

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Register expressions

See also

A register expression is a built-in assembler expression that consists solely of a register name.

Examples of register expressions are EAX, ECX, EDI, and ESI.

Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

See also

Built-in assembler register symbols

Immediate value expressions

Memory reference expressions

Memory reference expressions

See also

Memory references are built-in assembler expressions that denote memory locations.

Object Pascal labels, variables, typed constants, procedures, and functions belong to this category.

Memory references are further classified as either relocatable or absolute expressions.

See also

Register expressions

Immediate value expressions

Immediate value expressions

See also

Immediate values are built-in assembler expressions that are not registers and are not associated with memory locations.

This group includes Object Pascal untyped constants and type identifiers.

Immediate values are further classified as either relocatable or absolute expressions.

See also

Register expressions

Memory reference expressions

Relocatable vs. absolute expressions

See also

A relocatable expression denotes a value that requires relocation at link time. (Relocation is the process by which the linker assigns absolute addresses to symbols.)

An absolute expression denotes a value that requires no relocation.

Typically, an expression that refers to a label, variable, procedure, or function is relocatable, and an expression that operates solely on constants is absolute.

The absolute address of a label, variable, procedure, or function is not known until run time, when the code is loaded into memory at an arbitrary address.

The built-in assembler carries out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

See also

Register expressions

Memory reference expressions

Immediate value expressions

Built-in assembler register symbols

See also

The built-in assembler uses the following reserved symbols to denote CPU registers:

Symbols	Registers
EAX, EBX, ECX, EDX	32-bit general purpose
AX BX CX DX	16-bit general purpose
AL BL CL DL	8-bit low registers
AH BH CH DH	8-bit high registers
ESP, EBP, ESI, EDI	32-bit pointer or index
SP BP SI DI	16-bit pointer or index
CS DS SS ES	16-bit segment registers
ST	80x87 register stack

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands.

Register indexing

The base registers (BX and BP) and the index registers (SI and DI) can be written within brackets to indicate indexing. You can also index with all 32-bit registers.

These are the valid index register combinations:

```
[BP]
[BP+DI]
[BP+SI]
[BX]
[BX+DI]
[BX+SI]
[DI]
[SI]
[ESP]
[EAX+ECX]
[ESP+EAX-5]
```

See also

Register expressions

Built-in assembler expression symbols

You can use the built-in assembler to access almost all Object Pascal symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions.

The built-in assembler also provides several predefined type symbols.

Symbol	Value	Class	Type
Label	Address of label	Memory	Short
Constant	Value of constant	Immediate	0
Type	0	Memory	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory	Size of type
Procedure	Address of procedure	Memory	Near
Function	Address of function	Memory	Near
Unit	0	Immediate	0
@Code	Code segment address	Memory	0FFF0H
@Data	Data segment address	Memory	0FFF0H
@Result	Result var offset	Memory	Size of type

Symbols that cannot be used in built-in assembler expressions

The following symbols cannot be used in built-in assembler expressions:

- Standard procedures and functions (for example, Writeln, Chr)
- The special arrays Mem, MemW, MemL, Port and PortW
- String constants, floating-point constants, and set constants
- Labels that are not declared in the current block
- The @Result symbol outside a function

Built-in assembler local variables

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP.

The value of a local-variable symbol is its signed offset from EBP.

The built-in assembler automatically adds `[EBP]` in references to local variables.

Built-in assembler var parameter

The built-in assembler always treats a var parameter as a 32-bit pointer, and the size of a **var** parameter is always 4 (the size of a 32-bit pointer).

A const parameter is treated like a **var** parameter if the size of the parameter type is greater than 4 bytes.

In Object Pascal, the syntax for accessing a **var** parameter and a value parameter is the same. This is not the case in code you write for the built-in assembler.

Because **var** parameters are really pointers, you have to treat them as such. So, to access the contents of a **var** parameter, you first need to load the 32-bit pointer and then access the location to which it points.

Scope

A scope is provided by type, field, and variable symbols of a record or object type.

Like fully qualified identifiers in Object Pascal, a unit identifier opens the scope of a particular unit.

Operator

Some symbols, such as record types and variables, have a scope that can be accessed using the structure member selector (.) operator.

Type identifier

You can use a type identifier to construct variables as you write your program.

Each of these instructions generates the same machine code, which loads the contents of [EDI+4] into EAX:

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: Point;
  end;
asm
  MOV     EAX,(Rect PTR [EDI]).B.X
  MOV     EAX,Rect([EDI]).B.X
```

```
  MOV     EAX,Rect[EDI].B.X
  MOV     EAX,Rect[EDI].B.X
  MOV     EAX,[EDI].Rect.B.X
end;
```

ST(x) register symbol

The symbol ST represents the topmost register on the 8087 floating-point register stack.

Each of the eight floating-point registers can be referred to using `ST(x)`, where x is a constant between 0 and 7, indicating the distance from the top of the register stack.

Built-in assembler string constants

See also Example

In built-in assembler statements, string constants must be enclosed in single or double quotes.

To use a quotation mark in a constant, you need to use two consecutive quotation marks of the same type as the enclosing quotation marks. For example,

```
'"That''s all folks", he said.'
```

String constants of any length are allowed in DB directives, and cause, in the code segment; allocation of a sequence of bytes containing the ASCII values of the characters in the string.

When not in a DB directive, a string constant can be no longer than four characters, and denotes a numeric value that can be used as part of an expression.

The numeric value of a string constant is calculated as

```
Ord(Ch1)+ Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where

- Ch1 is the rightmost (last) character.
- Ch4 is the leftmost (first) character.

If the string is shorter than four characters, the leftmost (first) character(s) are assumed to be 0 (zero).

See also

Numeric constants

Examples

Here are some examples of string constants and their corresponding numeric values:

String constant	Hexadecimal value
`'a'`	00000061H
`'ba'`	00006261H
`'cba'`	00636261H
`'dcba'`	64636261H
`'a '`	00006120H
`'  a'`	20202061H
`'a'*2`	000000E2H
`'a'-'A'`	00000020H
`NOT 'a'`	FFFFFF9EH

Built-in assembler numeric constants

See also Example

Built-in assembler numeric constants must be integers between -2,147,483,648 and 4,294,967,295, and they must start with one of the digits 0 through 9 or a $ character.

By default, numeric constants use decimal (base 10) notation, but the built-in assembler supports binary (base 2), octal (base 8), and hexadecimal (base 16) notations as well.

To select this notation	Write a...
Binary	Letter `B` after the number
Octal	Letter `O` after the number
Hexadecimal	Letter `H` after the number, or a `$` before the number

Object Pascal expressions allow only decimal notation and hexadecimal notation (using a $ prefix); they do not support the B, O, and H suffixes.

When you write a hexadecimal constant using the H suffix, and the first significant digit is one of the hexadecimal digits A through F, an extra zero (0) in front of the number is required.

See also

String constants

Examples

`0BAD4H`	Hexadecimal constant
`$BAD4`	Hexadecimal constant
`BAD4H`	Identifier (it starts with a letter B, not a digit)

Built-in assembler reserved words

Within operands, the following reserved words have a predefined meaning to the built-in assembler:

Reserved words

AH
AL
AND
AX
BH
BL
BP
BX
BYTE
CH
CL
CS
CX
DH
DI
DL
DS
DWORD
DX
EAX
EBP
EBX
ECX
EDI
EDX
EID
ES
ESI
ESP
HIGH
LOW
MOD
NOT
OFFSET
OR
PTR
QWORD
SHL
SHR
SI
SP
SS
ST
TBYTE
TYPE
WORD
XOR

Reserved words always take precedence over user-defined identifiers. To access a user-defined symbol with the same name as a built-in assembler reserved word, you need to use the identifier override operator, (&). For example,

```
var
```

```
  ch: Char;
  ...
asm
  MOV &ch, 1
end;
```

Note: Avoid defining identifiers with the same names as built-in assembler reserved words, because such name confusion can lead to obsure and hard-to-find bugs.

Built-in assembler prefix opcodes

The built-in assembler supports the following prefix opcodes:

Opcode	What it means
LOCK	Bus lock
REP	Repeat string operation
REPE	Repeat string operation while equal
REPZ	Repeat string operation while 0
REPNE	Repeat string operation while not equal
REPNZ	Repeat string operation while not 0
SEGCS	CS (code segment) override
SEGDS	DS (data segment) override
SEGES	ES (extra segment) override
SEGSS	SS (stack segment) override

An assembler instruction can be prefixed by up to three of these opcodes.

If you specify a prefix opcode without an instruction opcode in the same statement, the prefix opcode affects the instruction opcode in the next assembler statement.

Because some 80x86 processors do not handle all combinations correctly, ordering in multiple prefix opcodes is important.

Built-in assembler instruction opcodes

See also

The built-in assembler supports all 8086/8087 and 80386/80387 instruction opcodes.

For complete descriptions of the instruction opcodes, refer to the Intel 80x86 and 80x87 reference manuals.

See also

Automatic jump optimization

Built-in assembler automatic jump optimization

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the most efficient form of a jump instruction.

When the target is a label (not a procedure or function), this automatic jump sizing applies to JMP and to all conditional jump instructions.

Opcode	Distance to target label	Built-in assembler generates
JMP	Within -128 to 127 bytes	Short jump
NOT	Within -127 to 128 bytes	Near jump
Conditional	Within -128 to 127 bytes	Short jump
Jumps	NOT within -127 to 128 bytes	Short inverse jump

Jumps to the entry points of procedures and functions are always near.

Built-in assembler DB, DW, and DD directives

Example

The built-in assembler supports three assembler directives: DB (define byte), DW (define word), and DD (define double word).

Each directive generates data corresponding to the comma-separated operands that follow that directive.

Dir	Operand type	Value range	Built-in assembler generates
DB	Constant expression	-128 to 255	1 byte
	Character string	Any length	Sequence of bytes corresponding to ASCII code of each character
DW	Constant expression	-32,768 to 65,535	1 word
	Address expression		Near pointer (offset word)
DD	Constant expression	-2,147,483,648 to 4,294,967,295	1 double word
	Address expression		Far pointer (offset word followed by segment word)

The data generated by the DB, DW, and DD directives is always stored in the code segment.

To generate uninitialized or initialized data in the data segment, use normal Pascal var or const declarations.

The only kind of symbol that can be defined in a built-in assembler statement is a label. All variables must be declared using Pascal syntax.

Example

Here are some examples of DB, DW, and DD directives:

Dir	Operand	Result
DB	0FFH	One byte
DB	0,99	Two bytes
DB	'A'	Ord('A')
DB	'Hello...', 0DH,0AH	String + CR/LF
DB	13,"Object Pascal"	Pascal-style string
DW	0FFFFH	One word
DW	0,9999	Two words
DW	'A'	Same as DB 'A',0
DW	'BA'	Same as DB 'A','B'
DW	MyVar	Offset of MyVar
DW	MyProc	Offset of MyProc
DD	0FFFFFFFFH	One double-word
DD	0,999999999	Two double-words
DD	'A'	Same as DB 'A',0,0,0
DD	'DCBA'	Same as DB 'A','B','C','D'
DD	MyVar	Pointer to MyVar
DD	MyProc	Pointer to MyProc

Built-in assembler statement syntax

The syntax of an assembler statement is

```
[Label:] <Prefix> [Opcode [Operand <,Operand>]]
```

where

- Label is a label identifier.
- Prefix is an assembler prefix opcode (operation code).
- Opcode is an assembler instruction opcode or directive.
- Operand is an assembler expression.

You can put multiple assembler statements on one line if they are separated by semicolons. A semicolon is not required between two assembler statements if they are on separate lines.

Comments must be written in Pascal style, using **{** and **}** or **(*** and ***)**.

Comments are allowed between assembler statements, but not within them.

Built-in assembler @Code, @Data, and @Result

In addition to providing access to almost all Pascal symbols, the built-in assembler implements the following special symbols:

Symbol	Meaning
@Code	The current code segment
@Data	The current data segment
@Result	The function result variable within the statement part of a function

Built-in assembler identifier override operator (&)

An identifier immediately following an ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved word.

```
&Identifier
```

Built-in assembler subexpression (...)

An expression within parentheses is a subexpression.

```
(Expression)
```

Subexpressions are evaluated completely before being treated as a single expression element.

Another expression can precede the subexpression. In this case, the result becomes the sum of the values of the two expressions, with the type of the first expression.

Built-in assembler memory reference operator [...]

An expression within brackets refers to a memory location.

```
[Expression]
```

This memory-reference expression is evaluated completely prior to being treated as a single-expression element.

To indicate CPU-register indexing, you can use the plus (+) operator to combine the memory-reference expression with the BX, BP, SI, or DI registers.

Another expression can precede the memory reference expression. In this case, the result becomes the sum of the values of the two expressions, with the type of the first expression.

Result

Always a memory reference.

Built-in assembler structure member operator (xxx . yyy)

A second expression is a member of the structure identified by the first expression.

```
Expression1.Expression2
```

Result

The sum of the two expressions.

Result type

The type of the second expression (`Expression2`).

Symbols belonging to the scope identified by the first expression can be accessed in the second expression.

Built-in assembler HIGH operator

HIGH returns the high-order 8 bits of the word-sized expression following the operator.

```
HIGH Expression
```

The expression must be an absolute immediate value.

Built-in assembler LOW operator

LOW returns the low-order 8 bits of the word-sized expression following the operator.

```
LOW Expression
```

The expression must be an absolute immediate value.

Built-in assembler unary plus (...)

Unary plus returns the expression following the plus with no changes.

```
+Expression
```

The expression must be an absolute immediate value.

Built-in assembler unary minus (-...)

Unary minus returns the negated value of the expression following the minus.

```
-Expression
```

The expression must be an absolute immediate value.

Built-in assembler segment override operator (: ...)

This operator instructs the assembler that the expression after the colon belongs to the segment specified by the segment register name before the colon (CS, DS, SS, FS, GS or ES).

```
XX:Expression
```

where

```
XX = CS, DS, SS, or ES.
```

Result

A memory reference with the value of the expression after the colon.

When a segment override is used in an instruction operand, the instruction will be prefixed by an appropriate segment override prefix instruction. This ensures that the indicated segment is selected.

Built-in assembler OFFSET operator

OFFSET returns the offset part (low-order dword) of the expression following the operator.

```
OFFSET Expression
```

Result

An immediate value.

Built-in assembler TYPE operator

TYPE returns the type (size in bytes) of the expression following the operator.

```
TYPE Expression
```

The type of an immediate value is 0.

Built-in assembler typecast operator (... PTR ...)

PTR casts the second expression to the type of the first expression.

```
Expression1 PTR Expression2
```

Result

A memory reference with the value of the second expression and the type of the first expression.

Built-in assembler multiplication operator (... * ...)

The * operator multiplies the first expression by the second expression.

```
Expression1 * Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler integer division operator (... / ...)

The / operator divides the first expression by the second expression and returns the integer part of the operation.

```
Expression1 / Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler modulus operator (... MOD ...)

MOD divides the first expression by the second expression and returns the remainder part of the operation.

```
Expression1 MOD Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler shift left operator (... SHL ...)

SHL shifts the first expression to the left by nnn bits, where nnn is the second expression.

```
Expression1 SHL Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler shift right operator (... SHR ...)

SHR shifts the first expression to the right by nnn bits, where nnn is the second expression.

```
Expression1 SHR Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler addition (... + ...)

The + operator adds the first expression to the second expression.

```
Expression1 + Expression2
```

The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value.

Result

A relocatable value if one of the expressions is a relocatable value; memory reference if either of the expressions is a memory reference.

Built-in assembler subtraction (... - ...)

The - operator subtracts the second expression from the first expression.

```
Expression1 - Expression2
```

The first expression can have any class, but the second expression must be an absolute immediate value.

Result

Has the same class as the first expression.

Built-in assembler bitwise negation (NOT)

NOT returns the bitwise negative (1's complement) of the expression after it.

```
NOT Expression
```

The expression must be an absolute immediate value.

Result

An absolute immediate value.

Built-in assembler bitwise AND

AND returns the bitwise AND of the two expressions.

```
Expression1 AND Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler bitwise OR

OR returns the bitwise OR of the two expressions.

```
Expression1 OR Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Built-in assembler bitwise exclusive OR (XOR)

XOR returns the bitwise exclusive OR of the two expressions.

```
Expression1 XOR Expression2
```

Both expressions must be absolute immediate values.

Result

An absolute immediate value.

Where built-in assembler functions return

Example

Functions using the built-in assembler directive **(asm)** must return their results as follows:

Result	Returned in	Type
Ordinal	AL (8-bit values)	Integer, Char, Boolean, and enumerated types
	AX (16-bit values)	
	EAX (32-bit values)	
Real	ST(0) on the coprocessor's register stack	Single, Double, Extended, Currency and Comp
Pointer	EAX	
String	Temporary location pointed to by @Result	

To modify the function result in ASM code blocks in normal functions, you must put the value in @Result. The epilog code for the Pascal function always copies from @Result to the registers listed above.

Example

```
function foo: Pointer
begin
  asm
    xor   eax,eax
    mov   @Result, Word[0], eax
    mov   @Result, Word[2], eax
  end;
end;
```

Short jump

1-byte opcode followed by 1-byte displacement.

Near jump

1-byte opcode followed by 2-byte displacement.

Short inverse jump

A short jump with the inverse condition jumps over a near jump to the target label (5 bytes in total).

The external assember

Use the external assembler to compile assembler statements, which can later be imported into your Object Pascal code.

For more information about the external assembler, choose from the following list of topics.

.MODEL and PROC directives

Linking external assembler code

Converting object files

External assembly language methods

.MODEL and PROC directives

See also Example

Turbo Assembler (TASM) makes it easy to program routines in assembly language and incorporate them in your Delphi applications. Turbo Assembler provides simplified segmentation and language support for Pascal programmers.

Directive	What it does
.MODEL	Specifies the memory model for an assembler module that uses simplified segmentation
PROC	Enables you to define your parameters in the same order as they are defined in your Pascal program

When you use the .MODEL directive, specifying the language PASCAL tells Turbo Assembler that arguments are pushed onto the stack from left to right.

When you use the PROC directive to define a function that returns a string, use the RETURNS option. The RETURNS option provides access to the temporary string pointer on the stack without affecting the number of parameter bytes added to the RET statement.

See also

Linking external assembler code

Example

Here's an example coded to use the .MODEL and PROC directives:

```
        .MODEL LARGE, PASCAL
        .CODE
MyProc  PROC   FAR I : BYTE, J : BYTE RETURNS Result : DWORD
        PUBLIC MyProc
        LES    DI, Result       ;get address of temporary string
        MOV    AL, I            ;get first parameter I
        MOV    BL, J            ;get second parameter J
      .
      .
      .
        RET
```

The Pascal function definition would look like this:

```
function MyProc(I, J: Char): string; external;
```

Linking external assembler code

See also

To link procedures and functions written in assembly language, use the $L compiler directive.

In order to use these assembly language procedures and functions, you need to declare them as external in your Delphi program or unit.

In the corresponding assembly language source file,

- All procedures and functions must be placed in a segment named CODE or CSEG, or in a segment whose name ends in _TEXT. The names of the external procedures and functions must appear in PUBLIC directives.
- Initialized variables can be declared in a segment named CONST or in a segment whose name ends in _DATA.
- Uninitialized variables can be declared in a segment named DATA or DSEG, or in a segment whose name ends in _BSS.

Uninitialized variables that reside in the same segment as the Object Pascal globals can be accessed using the DS segment register. Uninitialized variables declared elsewhere are private to the assembly language source file.

The EXTRN directive enables you to reference all procedures, functions, and variables declared in the Delphi program or unit, and those declared in the interface section of the used units.

See also

Assembly language methods

Converting object files

Converting object files

See also

When an object file appears in a $L directive, Object Pascal converts the file from the Intel relocatable object module format (.OBJ) to its own internal relocatable format.

This conversion is possible only if the following rules are observed:

- All procedures and functions must be placed in a segment named CODE or CSEG, or in a segment whose name ends in _TEXT.
- Initialized variables must be placed in a segment named CONST or in a segment whose name ends in _DATA.
- Uninitialized private variables must be placed in a segment named DATA or DSEG, or in a segment whose name ends in _BSS.
- All other segments are ignored, and so are GROUP directives.
- Code segments are always byte aligned.
- Data segments are always word aligned.
- Object Pascal ignores any data for segments other than the code segment (CODE, CSEG, or xxxx_TEXT) and the initialized data segment (CONST or xxxx_DATA). So, when declaring variables in the uninitialized data segment (DATA, DSEG, or xxxx_BSS), always use a question mark (?) to specify the value.
- Byte-sized references to EXTRN symbols are not allowed.

Segment definitions can optionally specify PUBLIC and a class name; however, both are ignored.

See also

The built-in assember

External assembly language methods

See also

Method implementations written in assembly language can be linked with Delphi programs using the $L compiler directive and the external reserved word.

The declaration of an external method in an object type is no different from that of a normal method.

The implementation of the method lists only the method header followed by the reserved word **external.**

In an assembly language source text, an at-sign (@) is used instead of a period (.) to write qualified identifiers (the period already has a different meaning in assembly language and cannot be part of an identifier).

The @ syntax can be used to declare both PUBLIC and EXTRN identifiers.

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 Delphi 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.

Delphi 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

Declaring interface routines

Reusing forms as 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 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

Declaring interface routines
Import units
Reusing forms as DLLs
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.

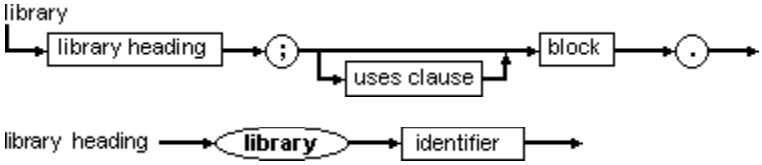
If you write your own DLLs, you must compile them separately.

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 Delphi 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 exampleimplements 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
Reusing forms as DLLs
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, Delphi 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
System unit
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 Delphi, 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 Delphi exception object.

If a DLL does not use the SysUtils unit, Delphi'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

See also 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 DELPHIMM.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 DELPHIMM.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.

See also
ShareMem unit

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

Delphi 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 Delphi 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 Exeption:

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

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 followng 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 unt.

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

Delphi'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

Delphi 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 Delphi 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 and 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

Changes to the Object Pascal language

This screen is intended to summarize the changes made in Delphi to the Object Pascal language for users of Borland Pascal 7.0. If you have not used Borland Pascal 7.0 and want to learn Object Pascal, see the Language definition.

Exception handling

Exception handling enables you to handle errors without exiting your application. For more information, see exception-handling.

New object-model

Delphi uses a new object model, however you can declare old-style objects in one unit, new-style objects in another, and use both of them in the same program. Keep in mind, however, that there are certain differences between the objects, and you are responsible for handling those disparities.

Note that new-style object declarations use the reserved word **class**, while old-style objects still use **object**.

The following changes are compatible with objects using the version 7.0 and earlier object model:

- Protected parts (in addition to public and private)
- Published parts

These changes apply only to new (version > 7.0) objects:

- Changes in Object declarations
- Changes in object use
- Properties
- Changes in method dispatching

Open-array construction

Open-array construction enables you to build an array and pass it as a parameter, all in one step. To construct an array parameter, you enclose the values for the array elements in square brackets. For more information, see Open-array parameters.

Type-safe open arrays containing multiple types

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.

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.

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.

Function result types

Functions can return any type, whether simple or complex, standard or user-defined, except old-style objects (as opposed to classes), and files of type text or **file of**. The only way to handle objects as function results is through object pointers.

Case statement optimizations

Two changes have been made to case statements.

- Ranges in case statements must not overlap.
- Placing case constants in ascending order allows the compiler to optimize the case into jumps instead of calculating each time.

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-}
```

You can set most switch directives from the Compiler Options page of the Options|Project dialog box.

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}
```

Parameter directives that allocate memory can be set on the Linker Options page of the Options|Project dialog box. Other parameter directives must be specified directly in code.

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 Delphi. From the list, you can jump to a complete definition of each directive.

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

$TYPEDADDRESS	Switch	Typed @ Operator
$TYPEINFO	Switch	Run-time Type Information
$U	Switch	Pentium-safe FDIV operations
$V	Switch	Var-String Checking
$VARSTRINGCHECKS	Switch	Var-String Checking
$W	Switch	Windows Stack Frame
$WARNINGS	Switch	Compiler Warnings
$WRITEABLECONST	Switch	Writeable typed constants
$X	Switch	Extended Syntax
$Y	Switch	Symbol Reference Information
$Z	Switch	Word Size Enumerated Types

Note: Most of these directives can be set from the Compiler Options page of the Options|Project dialog box.

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

Conditional symbol	Meaning
CONSOLE	Application is being compiled as a console application.
CPU386	CPU is an Intel 386 or better.
WIN32	The operating environment is the Win32 API.
VER90	Compiles based on the version specified. For example, code preceded by {$IFDEF 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

Delphi 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 typesthat 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

Note: You can also set the Word Align directive from the Compiler Options page of the Options|Project dialog box.

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 Delphi 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.

Note: You can also set the Complete Boolean Evaluation directive from the Compiler Options page of the Options|Project dialog box.

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. See Search|Find Error for information about this automatic error tracking.

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.

Note: You can also set the Debug Information directive from the Compiler page of the Project Options dialog box

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

ObjectBrowser

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. There are two possible results if a check fails:

- All run-time errors are translated into exceptions.
- 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.

Note: You can also set the Inupt/Output directive from the Compiler Options page of the Options|Project dialog box.

See also

$I 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.

Remarks

In previous versions of Delphi and Borland Pascal, typed constants were always writeable, corresponding to the {**$J+**} state. Old source code that uses writeable typed must be compiled in the {**$J+**} state, but for new applications it is recommended that you use initialized variables and compile your code in the default {**$J-**} state.

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 in the module's implementation part (not the interface part), and
- 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.

Note: You can also set the Local Symbol Information directive from the Compiler Options page of the Project Options dialog box.

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

Remarks

The **$P** directive is meaningful only for code compiled in the **{$H-}** state, and is provided for backward compatibility with earlier versions of Delphi and Borland Pascal.

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.

Note: You can also set the Open Parameters directive from the Compiler Options page of the Project Options dialog box.

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, there are two possible results:

- Run-time errors are translated into exceptions.
- 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.

Note: You can also set the Arithemitic Overflow Checking directive from the Compiler Options page of the Project Options dialog box.

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, there are the two possible results:

- Run-time errors are translated into exceptions.
- 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.

Note: You can also set the Range Checking directive from the Compiler Options page of the Options|Project dialog box.

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.

Note: You can also set the Stack Checking directive from the Compiler Options page of the Options|Project dialog box.

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.

Note: You can also set the Typed @ Operator directive from the Compiler Options page of the Options|Project dialog box.

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

Remarks

The **$V** directive is meaningful only for code that uses short strings, and is provided for backward compatibility with earlier versions of Delphi and Borland Pascal.

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.

Note: You can also set this option from the Compiler page of the Project Options dialog box.

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.

Note: You can also set the Windows Stack Frame directive from the Compiler page of the Project Options dialog box.

Extended syntax directive {$X }, { $EXTENDEDSYNTAX }

See also Compiler directives

This compiler directive enables or disables Delphi's extended syntax.

Syntax: {$X+} or {$X-}
{$EXTENDEDSYNTAX ON} or {$EXTENDEDSYNTAX OFF}

Default: {$X+}
{$EXTENDEDSYNTAX ON}

Scope: Global

On { $X+ }, { $EXTENDEDSYNTAX 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- }, { $EXTENDEDSYNTAX OFF }

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

Note: You can also set the Extended Syntax directive from the Compiler Options page of the Options|Project dialog box.

See also
SysUtils unit

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.

Note: You can also set the Symbol Information directive from the Compiler Options page of the Options|Project dialog box.

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, Delphi 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, Delphi 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 Classes unit of the Delphi 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.

Note For backwards compatibility with earlier versions of Delphi and Borland Pascal, the directives {**$Z-**} and {**$Z+**} are also supported. They correspond to {**$Z1**} and {**$Z4**} respectively.

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

Remarks

This directive corresponds to the Project|Options|Show Warnings option.

By placing code between **{$WARNINGS OFF}** and **{$WARNINGS ON}** directives, you can selectively turn off warnings you don't care about.

On { $WARNINGS ON }

When Warnings is on, the compiler generates warning messages in the Message Window when it detects unititialized 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 btwenne {$HINTS OFF} and {$HINTS ON} directives, you can selectively turn off hints that you don't care about.

This directive corresponds to the Project|Options|Show Hints option.

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+}, Delphi 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 Delphi's Object Pascal compiler are gauaranteed not to alter the meaning of a program. In other words, Delphi 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

Compiler error messages

Procedures and Functions (categorical)

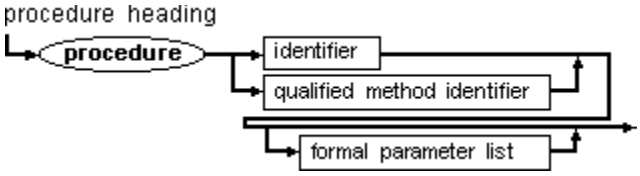
Reserved words

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:

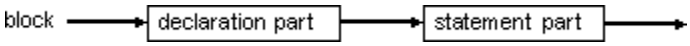
Shape	Represents
Box	Constructions
Circle	Reserved words, operators, and punctuation

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

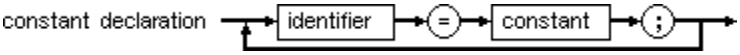
Block scope

Statements

Constant declarations

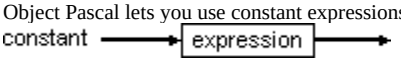
See also

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 Delphi 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

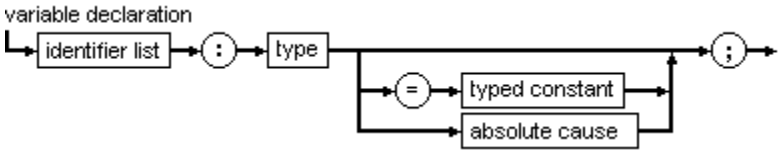
Typed constants

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.

Note Previous versions of Delphi and Borland Pascal supported a **$S** compiler directive to control stack-overflow checking. Stack-overflow checking is now completely automatic, but for backwards compatibility, the **$S** directive is still allowed.

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

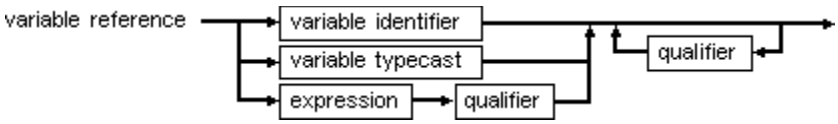
Variable references

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.

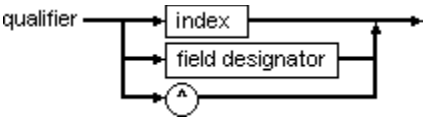
See also

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.

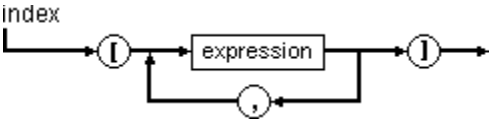
See also

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:

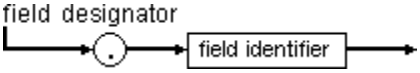
```
Memo1.Lines.Strings[0] := 'This is the first line.';
Memo1.Lines[0] := 'This is the first line.';
```

Field and object designators

See also Example Variable references

Field designators

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



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

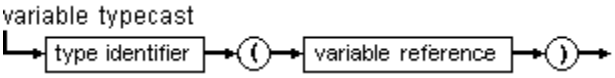
The following examples are references to dynamic variables.

```
Pl^
Pl^.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.



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

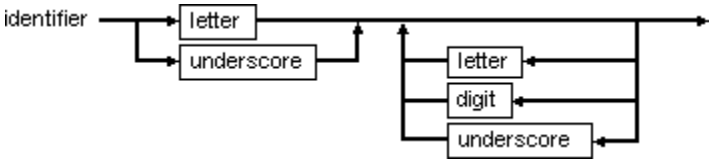
Procedures

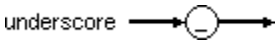
Programs

Types

Units

Variables





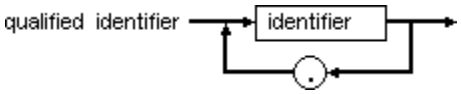
Identifiers are not case-sensitive.

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
```

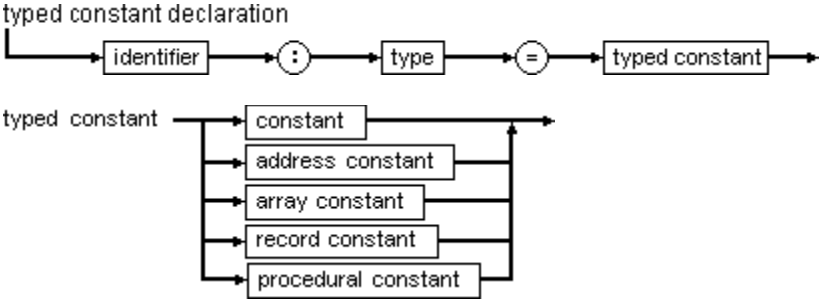
The following items are qualified identifiers.

```
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. For backwards compatibility with previous versions of Delphi and Borland Pascal, typed constants declared in the {**$J+**} state can be modified, and are in essence initialized variables. For new applications, use of the {**$J+**} state is not recommended.

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.

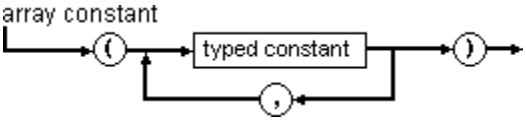
See also

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.



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

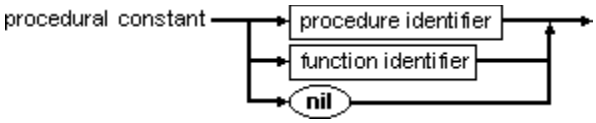
The following example declares pointer-type constants.

```
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

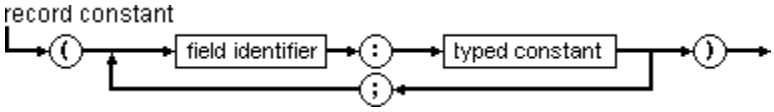
The following example assigns a procedure to a type constant.

```
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.



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 contants 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';
```

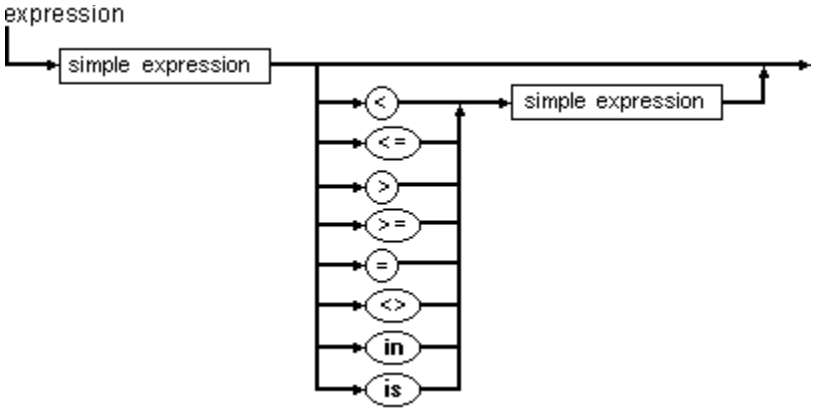
The following declares a short-string-type constant.

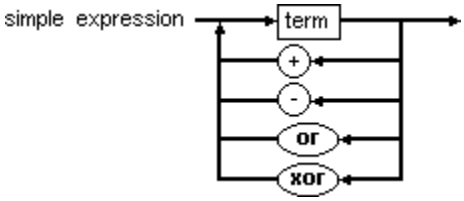
```
const
  ShortStr: string[5] = 'Short';.
```

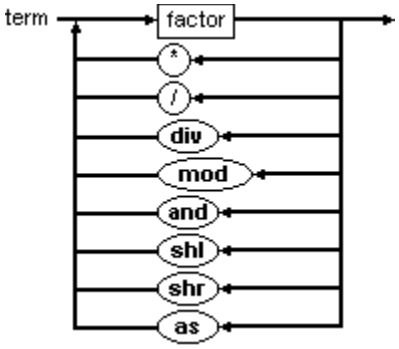
Expressions

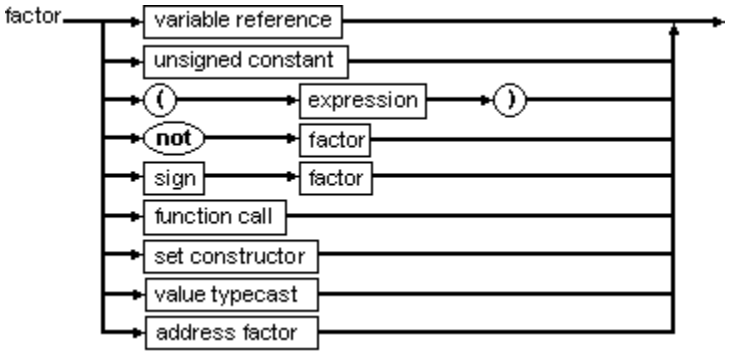
See also

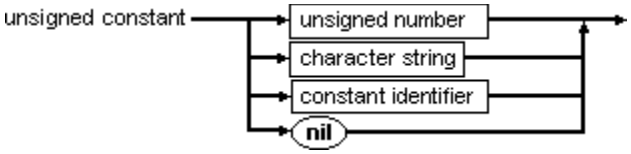
Expressions are a combination of operators and operands that evaluate to a single resulting value.











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
BASM expressions
Blocks
  Comments
Constant declarations
Function calls
Precedence of operators
Set types
Statements
Value typecasts
Variable reference

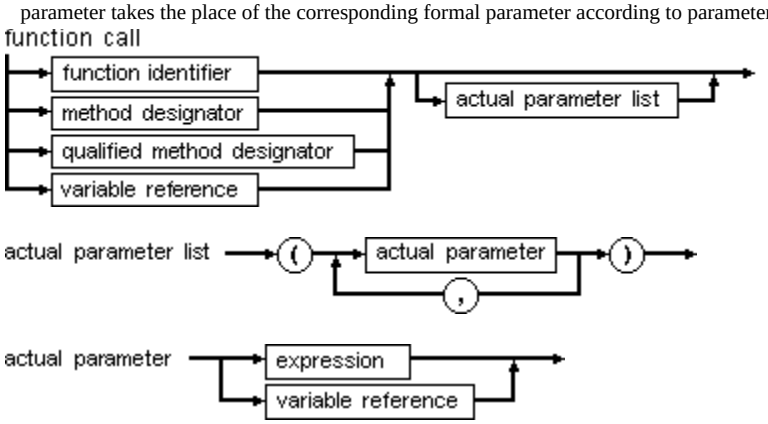
Function calls

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.



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

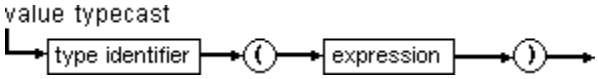
Parameters

Procedural types

Value typecasting

See also Example

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



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:

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

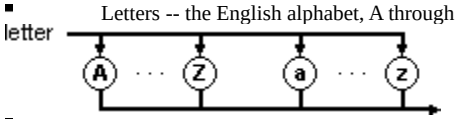
Some of the special symbols are operators.

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

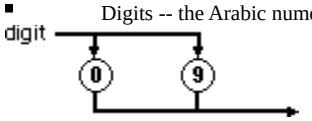
Character	Equivalent character pair
[	(.
]	).
{	(*
}	*)

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

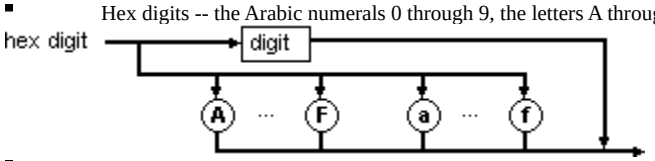
- Letters -- the English alphabet, A through Z and a through z



- Digits -- the Arabic numerals 0 through 9



- Hex digits -- the Arabic numerals 0 through 9, the letters A through F, and the letters a through f



- 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)

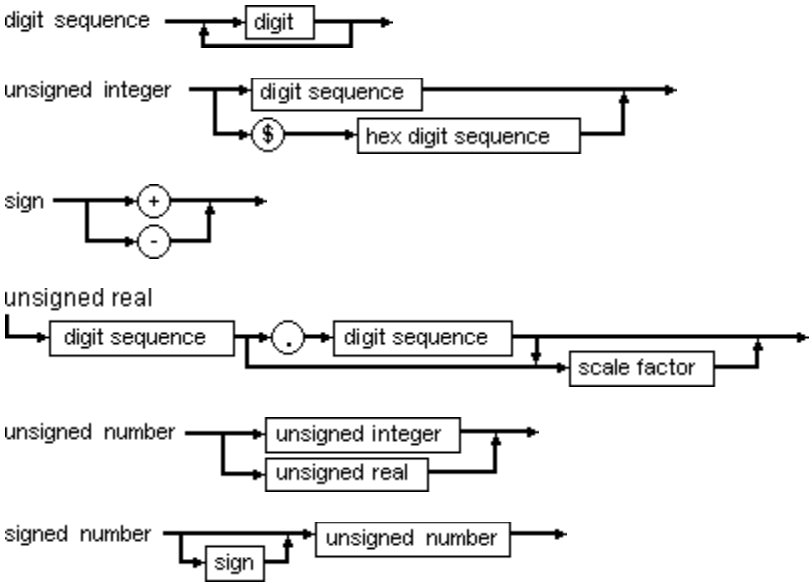
See also

Comments

Operators

Numbers

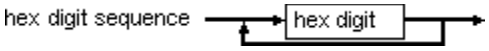
Ordinary decimal notation is used for numbers that are constants of integer and real types.



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 **x** 10-2

12.25e+6 or 12.25e6 both mean 12.25 **x** 10+6.

Character strings

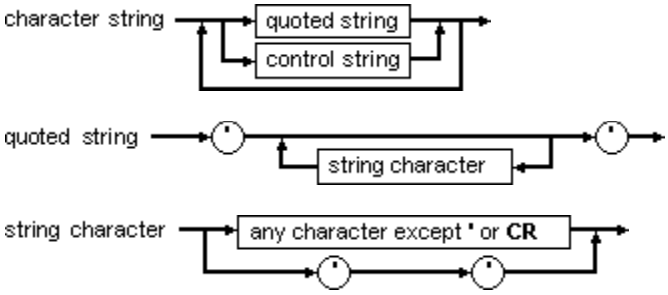
Example

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.

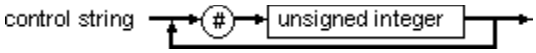
The length attribute of a character string is the actual number of characters within the apostrophes. The maximum length for a string is 255 characters.



Control characters

As an extension to standard Object Pascal, Delphi 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 {$X+} 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

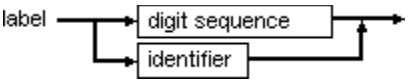
Comments

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

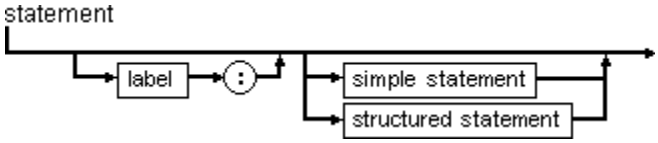
Blocks

Label (reserved word)

Statements

See also

Statements describe algorithmic actions that the program can execute.

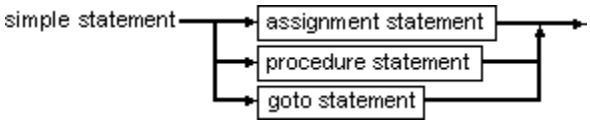


There are two basic types of statements:

- 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:

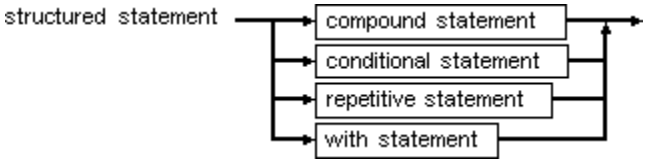
Assignment (:=) statements

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

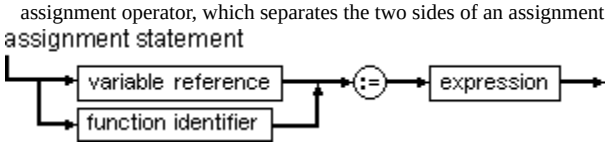
With statements

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 **:=**.



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 decendant 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

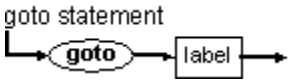
Examples

```
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

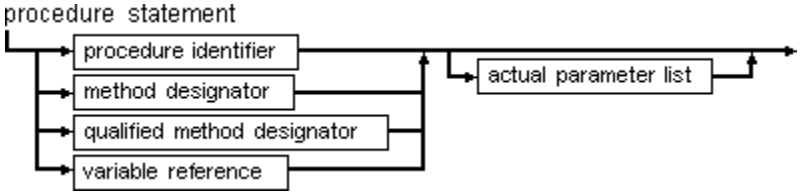
See also

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 pararmeters are passed to the formal parameters as part of the call.



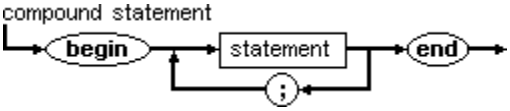
See also

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

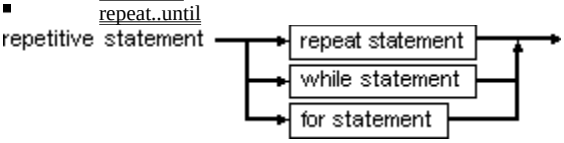
Loops

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

Loop	When to use
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

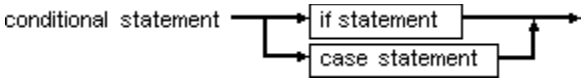
Conditional statements let you control whether certain expressions are evaluated. They test if a specific condition is met before executing the block of statements following the test.

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

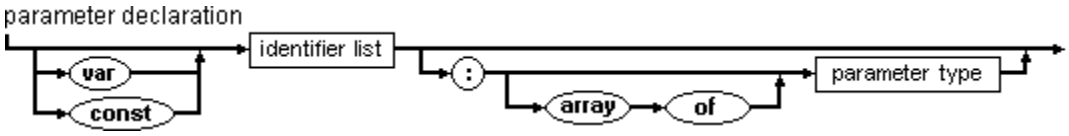
Relational operators

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
Low	Zero
High	The index of the last element in the actual array parameter
SizeOf	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

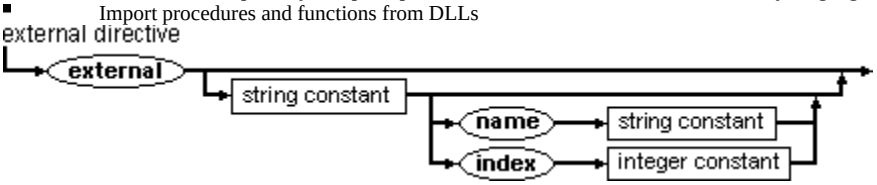
TVarRec type

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 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 Delphi program provides the "make" logic information traditionally found in make or project files of other languages. With the **uses** clause, Delphi can build all the dependency information into the module itself and reduce the chance for error.

Delphi 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 Delphi 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, Delphi 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

Delphi 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 Delphi 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 (Delphi 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

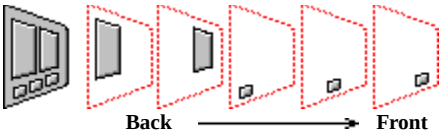
Exception handling

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**	Second	Multiplicative and type casting
and shl shr		operators
+ - or xor	Third	Additive operators
= <> < >	Fourth (low)	Relational, set membership,
<= >= **in is**		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
Run-time type 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 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 operator 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 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 mod J = I - (I 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

Run-time type information

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

Run-time type information
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 shl J` and `I 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 throught the $B compiler directive. In the defualt state **{$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

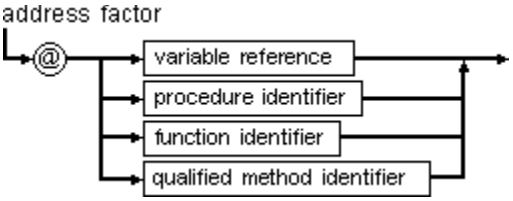
Boolean operators

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.

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

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
Assembler statement
Comments
External

Assembler

See also Standard directives

Syntax

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 enty 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

@Result

Asm

Built-in assembler entry and exit code

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 Delphi, see the Delphi 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.

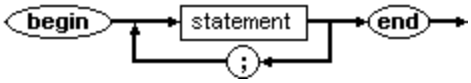
See also

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;
```

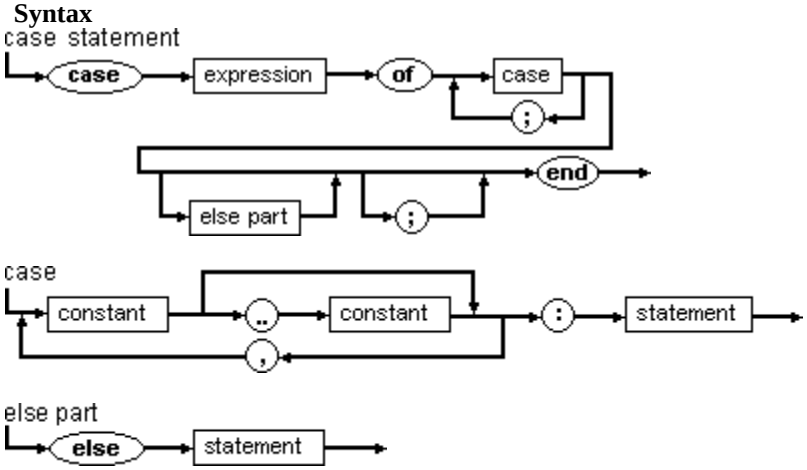
See also

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

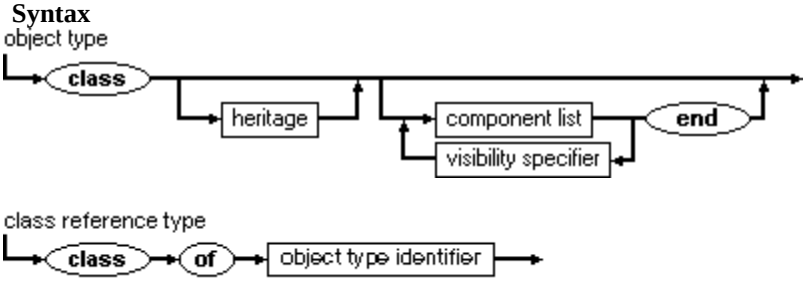
register

stdcall

Class

See also Reserved words

Syntax



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 references

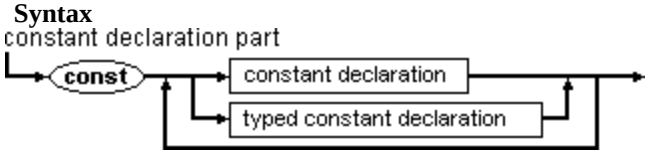
Object-type scope

Object types

Const

See also Example Reserved words

Syntax



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.

Delphi 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

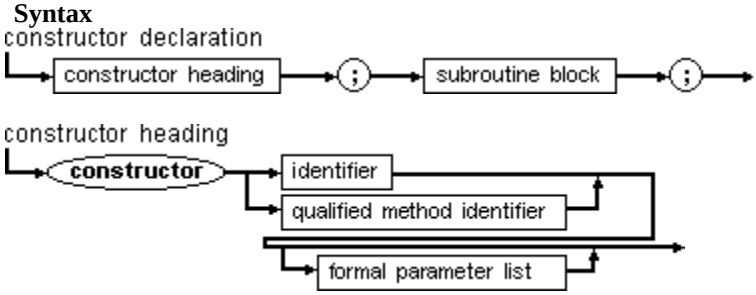
Examples

```
(* 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



Description

A constructor defines the actions associated with creating an object. It must be declared using the reserved word **constructor.** All Delphi 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.

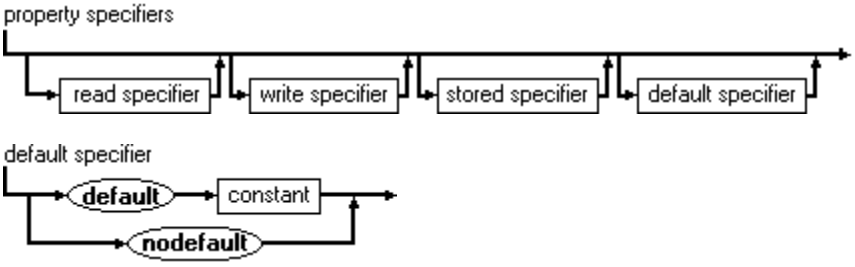
See also

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

Default array properties

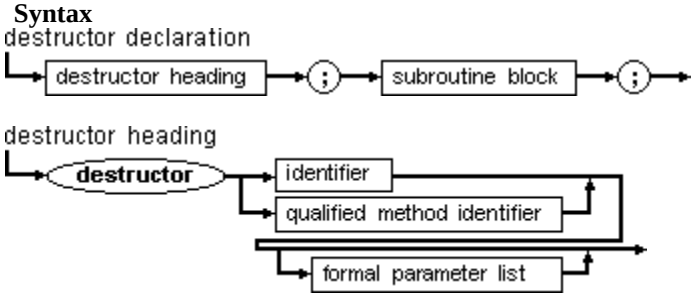
Nodefault

Storage specifiers

Destructor

See also Reserved words

Syntax



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

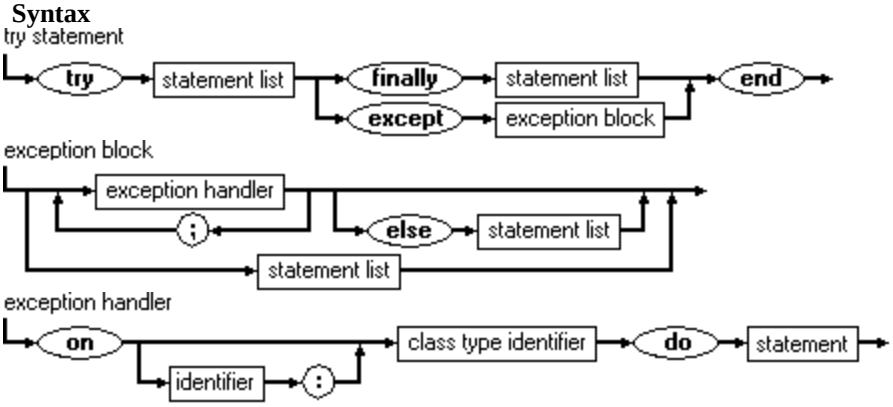
Examples

```
(* 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



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

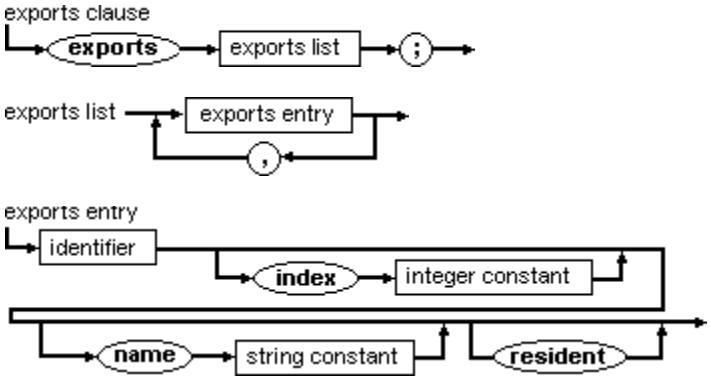
Finally

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

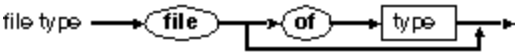
Functions

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;
```

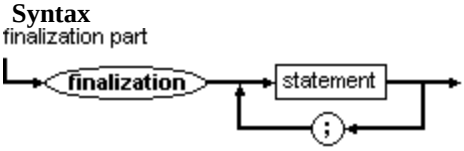
See also

Of

Finalization

See also Reserved words

Syntax



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 unics 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 ot execute, the coresponding 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.

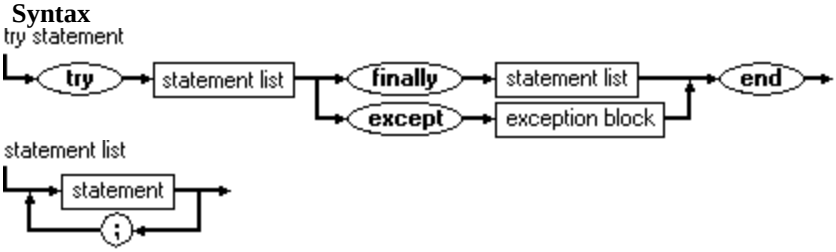
See also

Interface
Implementation
Initialization
Units

Finally

See also Reserved words

Syntax



Description

The **finally** reserved word marks the section of a protected block that always executes, even if an execption 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

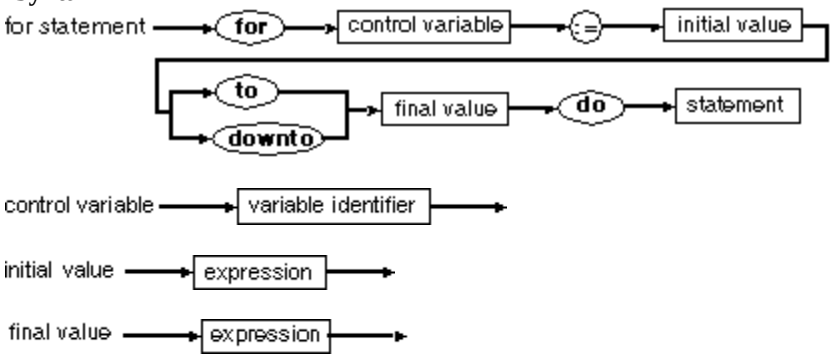
Try

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

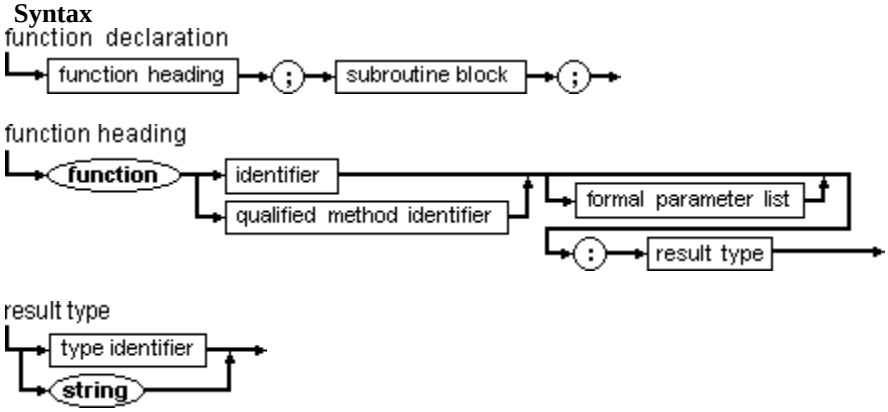
Assembler

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

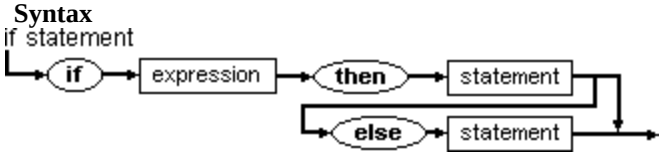
Example

```
label 1, 2;
goto 1
   .
   .
   .
1: WriteLn ('Abnormal program termination');
2: WriteLn ('Normal program termination');
```

If ... Then ... Else

See also Example Reserved words

Syntax



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

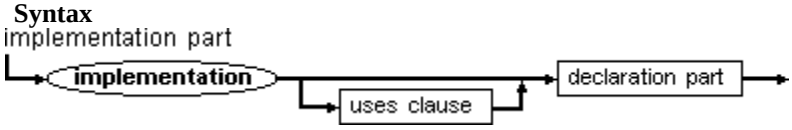
```
(* if statements *)
if (I < Min) or (I > Max) then I := 0;
if x < 1.5 then
  z := x + y
else
  z := 1.5;
```

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



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 athe following:

- Identical to the declaration in the interface
- In the short form

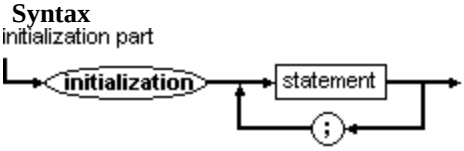
See also

$L filename
Interface
Initialization
Finalization
Units

Initialization

See also Reserved words

Syntax



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.

For information about using indexes with properties, see Index specifiers.

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

Calling inherited message handlers

Object types

Inline

Reserved words

Description

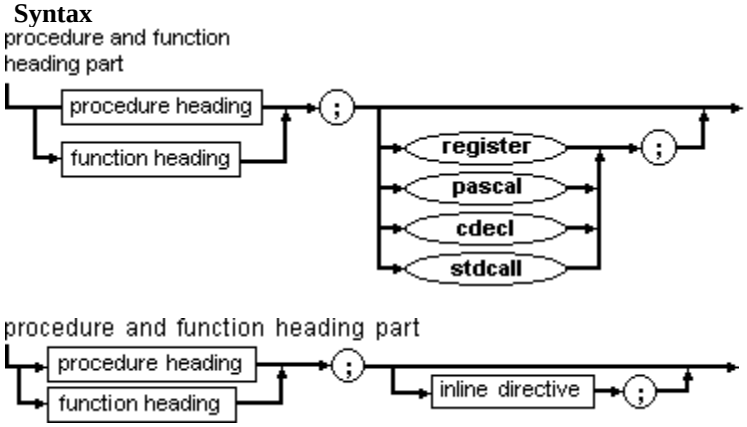
The reserved word **inline** is not used in this version of Object Pascal. It remains reserved, however, for future use.

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



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

Storage specifiers

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
Class
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 Delphi, 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.

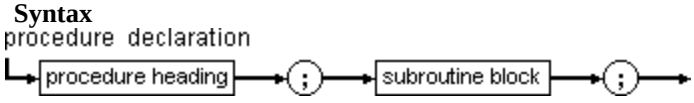
See also

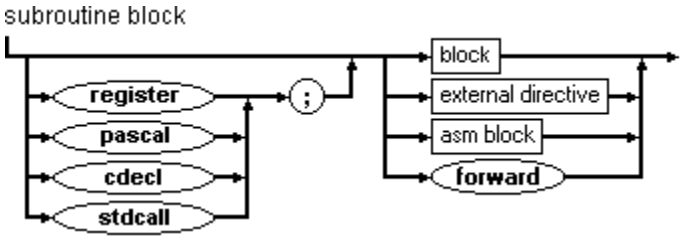
Objects
Private parts
Protected
Public
Rules of scope
Object-type scope
Units

Procedure

See also Example Reserved words

Syntax





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;
```

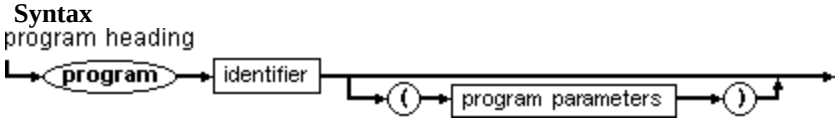
See also

Functions
Parameters
Procedure statements
Procedural-type constants
Calling conventions

Program

See also Reserved words

Syntax



Description

The reserved word **program** is placed at the top of a program and specifies the program's name.

See also

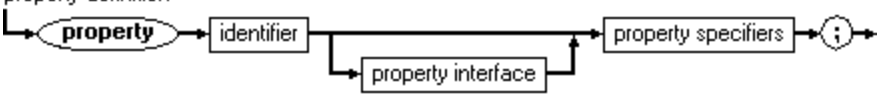
Uses clause
Labels
Constants
Types
Variables
Procedures
Functions
Statements

Property

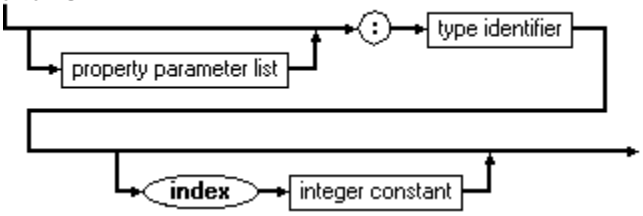
See also

Syntax

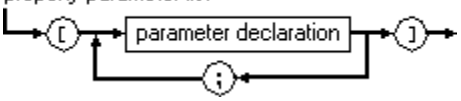
property definition



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

Properties

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 Delphi Object Inspector uses the published interface of objects in the Component palette to determine the properties and events it displays.

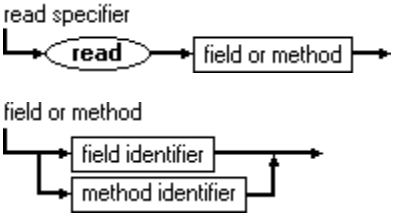
See also

Component visibility
Private
Protected
Public
Published parts
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

Access methods

Properties

Write

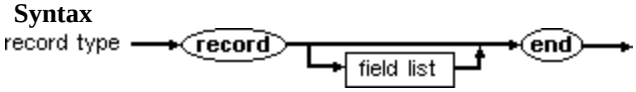
Example

```
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

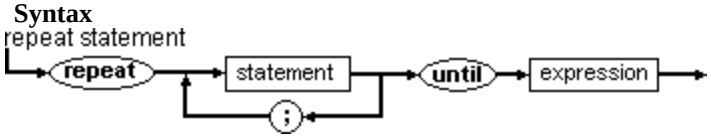
pascal

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.

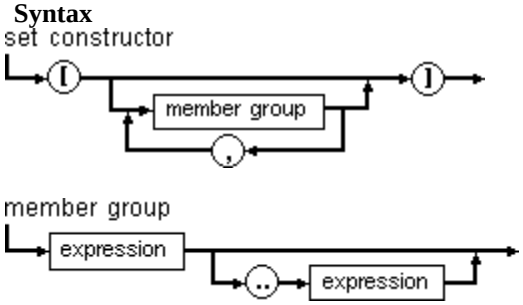
Self can be used as a pointer to the instance through which the method is being called.

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



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

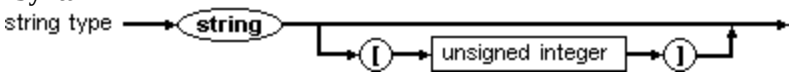
pascal

register

String

See also Reserved words

Syntax



Description

The reserved word **string** is used to declare string type variables.

See also

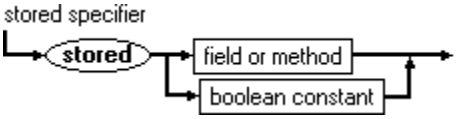
Indexes

String types

Stored

See also 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.

See also

Storage specifiers

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;
```

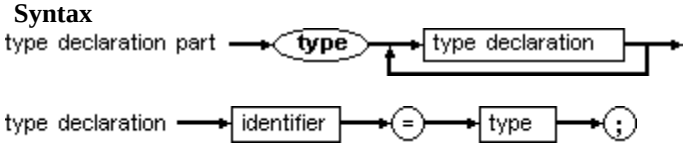
See also

Exception handling
Protecting resource allocations

Type

See also Reserved words

Syntax



Description

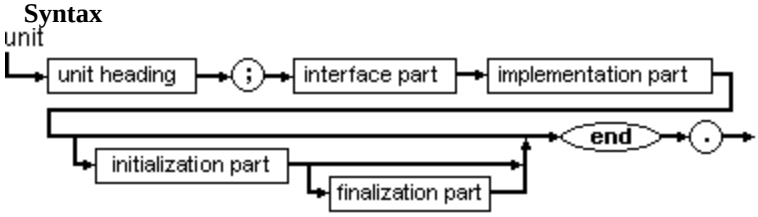
A **type** declaration specifies an identifier that denotes a type. A variable's **type** defines the set of values it can have and the operations that can be performed on it.

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

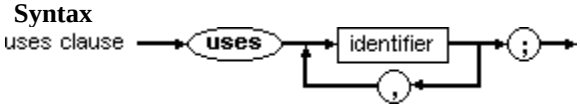
Standard units

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.

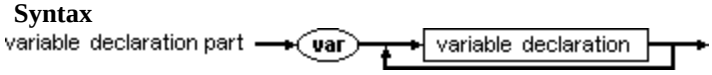
Example

```
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**.

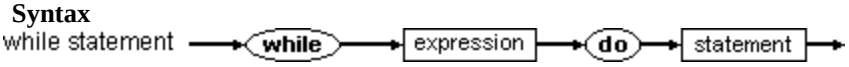
See also

Dynamic methods
Objects
Self
Virtual methods

While

See also Example Reserved words

Syntax



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)

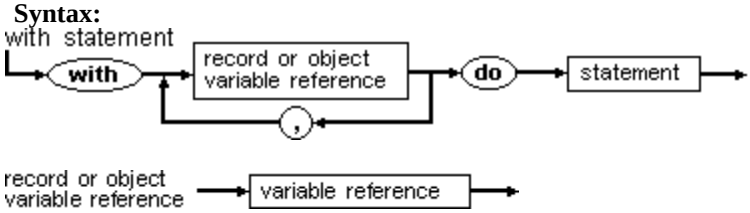
Example

```
{ 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

Access methods

Properties

Read

Delphi Units

The following is a listing of the standard units that are shipped with Delphi.
See the Delphi Library Reference online Help for detailed information on each unit.

Buttons Unit
Classes Unit
ClipBrd Unit
Controls Unit
DB Unit
DBCtrls Unit
DBGrids Unit
DBLookup Unit
DBTables Unit
DDEMan Unit
Dialogs Unit
DsgnIntf Unit
ExtCtrls Unit
FileCtrl Unit
Forms Unit
Graphics Unit
Grids Unit
IniFiles Unit
Mask Unit
Menus Unit
Messages Unit
MPlayer Unit
Outline Unit
Printers Unit
Report Unit
StdCtrls Unit
System Unit
SysUtils Unit
TabNotBk Unit
Tabs Unit
TOCtrl Unit
WinCrt Unit
WinProcs Unit
WinTypes Unit

Type declarations

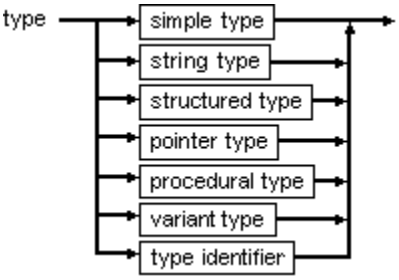
See also Language definition

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 diveded 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.

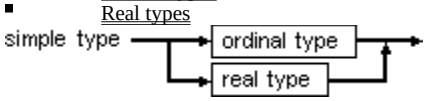
Applications should use the generic formats whenever possible, since they generally result in the best performance for the underlying CPU and operating system. The fundamental types should be used only when the actual range and / or storage format matters to the application.

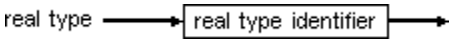
Simple types

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.

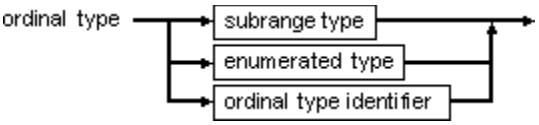
Comparing simple types

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.



Delphi 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 {$R+}, Pred produces a run-time error that you can handle by using exceptions.
Succ	Returns the successor of the value. If the value in question is the last value in the ordinal type and if range checking is enabled {$R+}, 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:

- False < True
- Ord(False) = 0
- Ord(True) = 1
- Succ(False) = True
- Pred(True) = 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 expectd, 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

Chr

Ord

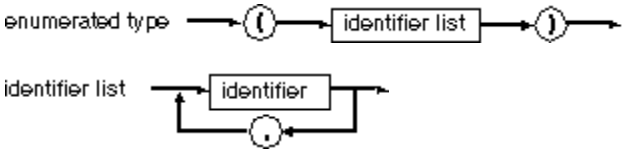
Type compatibility

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

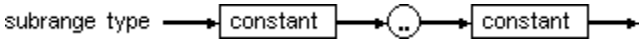
Enumerated type example

```
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.

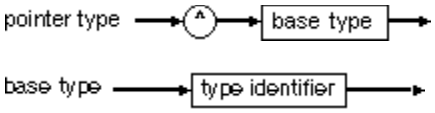
The Currency type is a fixed-point data type suitable for monetary calculations. It is stored as a scaled 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Note The Real type is provided for backward compatibility with earlier versions of Delphi and Borland Pascal. Since the storage format of the Real type is not native to the Intel CPU family, operations on Real type values are slower than the other floating-point types.

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
@ operator	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

Delphi 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.

Earlier versions of Delphi allowed allowed usage such as the following:

```
type
  PointerType = array[0..5] of ^Undefined;
  Undefined = array[0..5] of Longint;
```

That syntax is not supported by the current compiler.

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 declared in Delphi 2.0 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.
PShortString	Points to a variable of type ShortString. Mainly used when porting Delphi 1.0 16-bit code that uses PString to Delphi 2.0.
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.

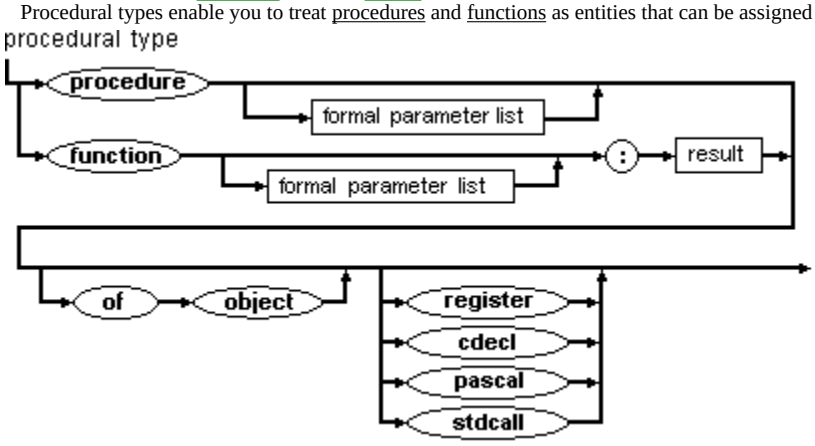
See also

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.



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 catagories 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 of function, and is encoded as a pointer that stores the address of a global procedure or function.

See also

Method pointers

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

Delphi 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 short string type primarily exists for reasons of backward compatibility with earlier versions of Delphi and Borland Pascal.

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

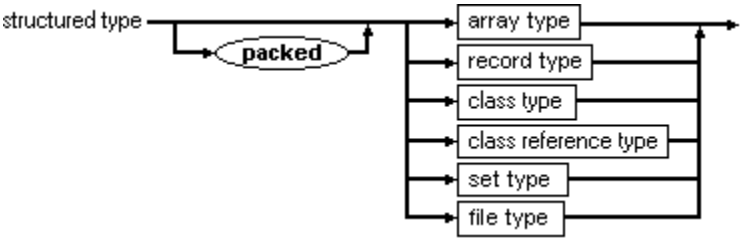
Short string types

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.



Delphi'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

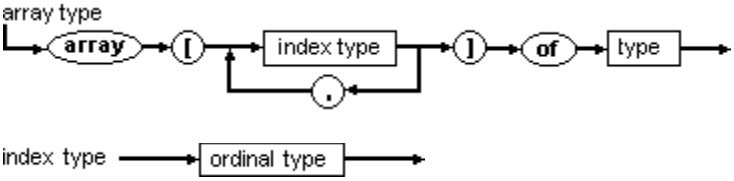
Structured-type constants

Type compatibility

Array types

See also Example Structured types

Arrays are one dimensional or multidimensional containers that hold multilple 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 type 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 }
```

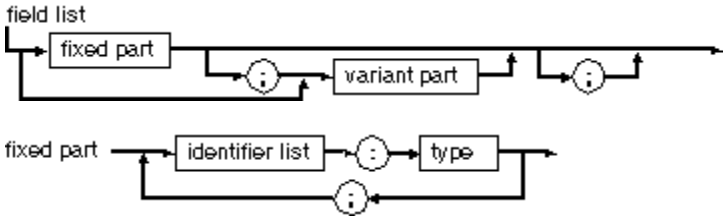
See also

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.

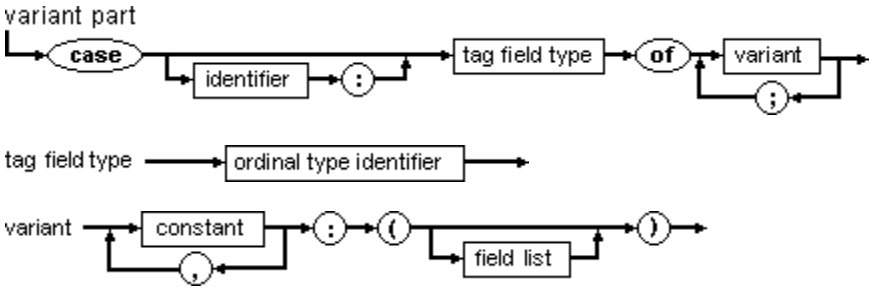


The field name and field type must be assigned in the record-type declaration.

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

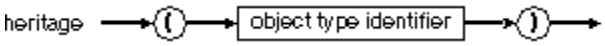
A class type is a structure consisting of a fixed number components.

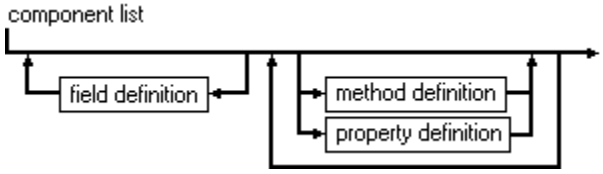
Possible components of a class are

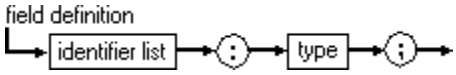
- fields
- methods
- properties

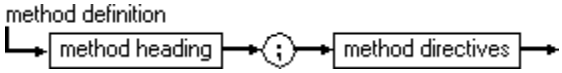
Unlike other types, a class type can be declared only in a type declaration part in the outermost scope of a program or unit. Therefore, a class type cannot be declared in a variable declaration part or within a procedure, function, or method block.

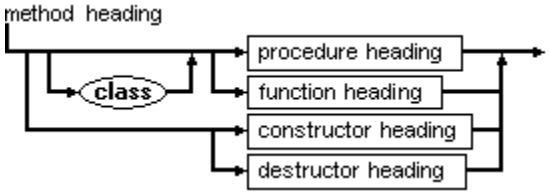
A class type is declared using the reserved word **class** and defines the contents of a class. Classes are also called "object types." The two terms are interchangeable.

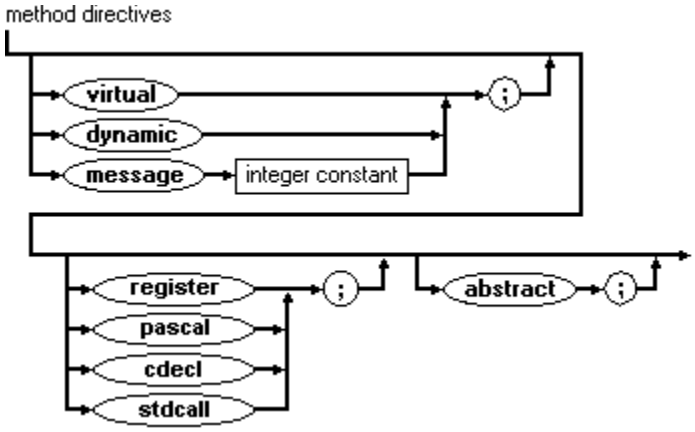












You must globally declare a class type. A class type cannot be declared in a variable declaration part or within a procedure, function, or method block.

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 an 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 assigment-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)
Class methods
Component scope
Component visiblity
Default ancestor
Forward class declaration
Instantiating objects
Methods
TObject

Class type example

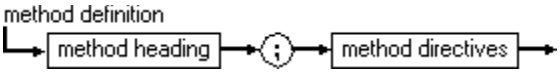
The following example declares the class TField

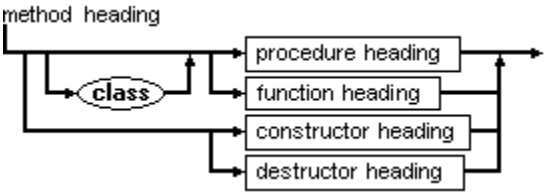
```
TField = class
private
  X, Y, Len: Integer;
  FName: string;
public
  constructor Copy(F: TField);
  constructor Create(FX, FY, FLen: Integer; FName: string);
  destructor destroy; override;
  procedure Display; virtual;
  procedure Edit; dynamic;
protected
  function GetStr: string; virtual;
  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;
```

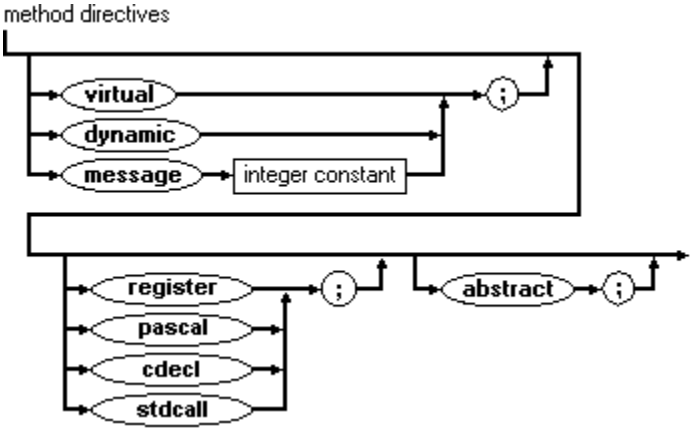
Methods

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

Dynamic methods

Class methods

Override directive

Message handling methods

Abstract 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
Method pointers
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

Dynamic methods
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. Analagous 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.

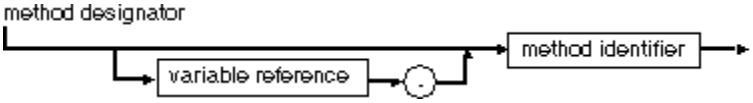
See also

Constructors and destructors
Dynamic methods
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

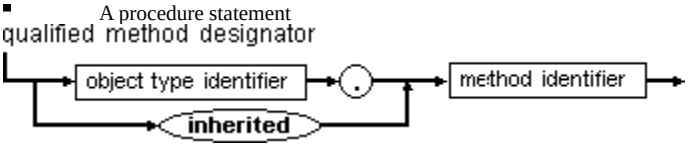
Function calls
Method declarations
Parameters
Procedure statements
Qualified-method activation
With statement

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



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
Overriding methods
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

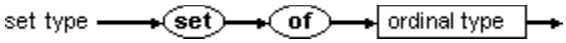
Component identifiers declared in the component list that immediately follows the object type heading have the published visibility attribute if the object type is compiled in the {**$M+**} state or is derived from a class that was compiled in the {**$M+**} state. Otherwise, such component identifiers 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 comparisions 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 assigment 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 T .
- 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 descandant 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
  TCRef: TComponentRef;
  NewComponent: TComponent;

...

TCRef := TButton; {TCRef can now be used anywhere the name of the
                   class TButton could be used}
NewComponent := TCRef.Create; {TCRef is used here to create a new
                               TButton}

...
```

Null-terminated strings

See also

Delphi 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 Delphi'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^`. 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

See also 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.

See also
SysUtils unit

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.

Delphi character sets

Delphi 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 suppored 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 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

See also 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 propogate 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 propogate 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 Delphi 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.

See also

VarType function

TVarData type

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 assigment 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 varSmallint varInteger varError	Converts one integer format to another, raises an EVariantError exception if value does not fit in destination format.
varSingle varDouble varCurrency	Rounds real value to nearest integer, raises an EVariantError exception if result does not fit in destination format.
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 varString	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.
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 varSmallint varInteger varError	Converts integer to real.
varSingle varDouble varCurrency	Converts from one real format to another, raises an EVariantError exception if value does not fit in destination format.
varDate	Interprets value as a Double, converts value to destination format, raises an EVariantError exception if value does not fit in destination format.
varOleStr varString	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.
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 varSmallint varInteger varError	Converts integer to Double, interprets result as a date and time value.
varSingle varDouble varCurrency	Converts value to Double, interprets result as a date and time value.
varDate	No translation.
varOleStr varString	Converts string to a date and time value using the Regional Settings in the Windows Control Panel.
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 varSmallint varInteger varError	Converts integer value to its string representation.
varSingle varDouble varCurrency	Converts real value to its string representation using the Regional Settings in the Windows Control Panel.
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 varSmallint varInteger varError	False if value is zero, True if value is non-zero.
varSingle varDouble varCurrency	False if value is zero, True if value is non-zero.
varDate	Interprets value as a Double, returns False if value is zero, True if value is non-zero.
varOleStr varString	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.
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 /, 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 / operator, the operation is performed as a Double operation.
Double	For the +, –, *, and / 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 +, –, *, and / 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 + 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%LockingVariantArraysSAExample%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;
```

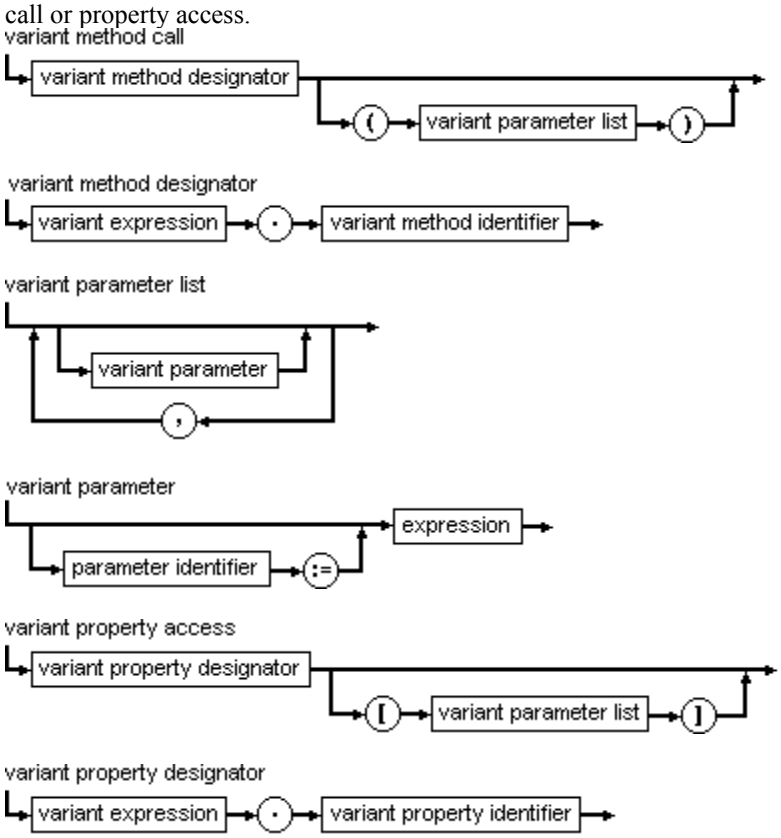
See also
Variant array dimensions

Variants and OLE Automation objects

See also Example Variant types

When a variant contains a reference to an OLE Automation object, Delphi 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.



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 Delphi 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 succesfully 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

Compiler messages

Run-time error messages

The following list shows all the error, warning, and hint messages generated by the Delphi compiler, in alphabetical order. To see further information on a particular message, click its text in the list.

Many of the messages contain variable text, so it might be difficult to locate a particular message in the list.

Note The most convenient way to get information on a message you receive in the Integrated Development Environment (IDE) is to highlight the message in the message window and press F1.

Declaration of <name> differs from previous declaration
Default property must be an array property
Default values must be of ordinal, pointer or small set type
Destination cannot be assigned to
Destination is inaccessible
Dispid '<number>' already used by '<name>'
Dispid clause only allowed in OLE automation section
Division by zero
Division by zero
Duplicate case label
Duplicate message method index
Duplicate tag value
Dynamic method or message handler not allowed here
Dynamic methods and message handlers not allowed in OLE automation section
Element 0 inaccessible - use 'Length' or 'SetLength'
Error in numeric constant
EXCEPT or FINALLY expected
EXPORTS allowed only at global scope
Expression has no value
Expression is not a procedure
Expression too complicated
Field definition not allowed in OLE automation section
Field definition not allowed after methods or properties
Field or method identifier expected
File not found: <Filename>
File type not allowed here
Fn requires 2 <= n <= 18
For loop control variable must be simple local variable
For loop control variable must have ordinal type
FOR or WHILE loop executes zero times - deleted
FOR-Loop variable '<name>' cannot be passed as var parameter
FOR-Loop variable '<name>' may be undefined after loop
Format specifier must be C, S, D, H, X, Fn, P, R or nM
Function needs result type
'GOTO <label>' leads into or out of TRY statement
Identifier redeclared: '<name>'
Illegal character in input file: '<Char>' (<Hexadecimal value>)
Illegal message method index
Illegal reference to symbol '<name>' in object file '<Filename>'
Illegal type in OLE automation section: '<typename>'
Illegal type in Read/Readln statement
Illegal type in Write/Writeln statement
Inaccessible value
Incompatible format specification
Incompatible types: '<name>' and '<name>'
Incompatible types: <text>
Incompatible types
Inline assembler stack overflow
Inline assembler syntax error
Instance variable '<name>' inaccessible here
Integer constant or variable name expected
Integer constant too large
Internal error: <ErrorCode>
Invalid combination of opcode and operands
Invalid compiler directive: <Directive>
Invalid function result type

Invalid message parameter list
Invalid register combination
Invalid typecast
Label '<name>' is not declared in current procedure
Label already defined: '<Labelname>'
Label declaration not allowed in interface part
Label declared and referenced, but not set: '<label>'
Label expected
Left side cannot be assigned to
Line too long (more than 255 characters)
Local class or object types not allowed
Local procedure/function '<name>' assigned to procedure variable
LOOP/JCXZ distance out of range
Low bound exceeds high bound
Memory reference expected
Method '<name>' hides virtual method of base type '<name>'
Method '<name>' not found in base class
Method identifier expected
Missing ENDIF directive
Missing operator or semicolon
Missing or invalid conditional symbol in '$<symbol>' directive
Missing parameter type
Necessary library helper function was eliminated by linker
No definition for abstract method '<name>' allowed
No source line for this procedure
Not enough actual parameters
Number of elements differs from declaration
Numeric overflow
Object or class type required
Object type required
Only register calling convention allowed in OLE automation section
Operand size mismatch
Operator not applicable to this operand type
Order of fields in record constant differs from declaration
Ordinal type required
Out of memory
Overflow in conversion or arithmetic operation
Overriding automated virtual method '<name>' cannot specify a dispid
PACKED not allowed here
Pointer type required
Procedure cannot have a result type
Procedure DISPOSE needs destructor
Procedure FAIL only allowed in constructor
Procedure NEW needs constructor
PROCEDURE or FUNCTION expected
Procedure or function name expected
Program or unit recursively uses itself
Property '<name>' does not exist in base class
Published property '<name>' cannot be of type <type>
Published Real property '<name>' must be Single, Double or Extended
Re-raising an exception only allowed in exception handler
Read error on <Filename>
Record, object or class type required
Redeclaration of '<name>' hides a member in the base class
Redeclaration of property not allowed in OLE automation section
Return value of function '<Functionname>' might be undefined

Seek error on <Filename>
Segment/Offset pairs not supported in Borland 32-bit Pascal
Sets may have at most 256 elements
Size of published set '<name>' is >4 bytes
Slice standard function only allowed as open array argument
Statement expected, but expression of type '<type>' found
Statements not allowed in interface part
String constant too long
String constant truncated to fit STRING[<number>]
Strings may have at most 255 elements
Structure field identifier expected
Syntax error in real number
System unit out of date or corrupted: missing '<name>'
Text after final 'END.' - ignored by compiler
This form of method call only allowed for class methods
This form of method call only allowed in methods of derived types
This type cannot be initialized
Thread local variables cannot be ABSOLUTE
Thread local variables cannot be local to a function
Too many actual parameters
Too many conditional symbols
Type '<name>' has no type info
Type '<name>' is not yet completely defined
Type '<name>' must be a class to have a PUBLISHED section
Type '<name>' must be a class to have OLE automation
Type '<name>' needs finalization - not allowed in file type
Type '<name>' needs finalization - not allowed in variant record
Type expected
Type not allowed in OLE Automation call
Type of expression must be BOOLEAN
Type of expression must be INTEGER
TYPEINFO standard function expects a type identifier
TYPEOF can only be applied to object types with a VMT
Types of actual and formal var parameters must be identical
Undeclared identifier: '<name>'
Unexpected end of file in comment started on line <Number>
Unit <Unit1> was compiled with a different version of <Unit2>
Unit name mismatch: '<Unitname>'
Unknown directive: '<Directive>'
Unnamed arguments must precede named arguments in OLE Automation call
Unsatisfied forward or external declaration: '<Procedurename>'
Unterminated string
Value assigned to '<name>' never used
Variable '<name>' inaccessible here due to optimization
Variable '<name>' is declared but never used in '<name>'
Variable '<name>' might not have been initialized
Variable required
Virtual constructors are not allowed
Write error on <Filename>
Wrong or corrupted version of RLINK32.DLL

"Ordinal type required"

See also ExampleCompiler error messages

Description

The compiler required an ordinal type at this point.

Ordinal types are the predefined types Integer, Char, WideChar, Boolean, and declared enumerated types. Ordinal types are required in several different situations:

The index type of an array must be ordinal.

- The low and high bounds of a subrange type must be constant expressions of ordinal type.
- The element type of a set must be an ordinal type.
- The selection expression of a case statement must be of ordinal type.
- The first argument to the standard procedures Inc and Dec must be a variable of either ordinal or pointer type.

Examples

{ The index type of an array must be an ordinal type - type TByteSet is a set, not an ordinal. }

```
program Produce;
type
  TByteSet = set of 0..7;
var
  BitCount: array[TByteSet] of Integer;
begin
end.
```

{ Supply an ordinal type as the array index type. }

```
program Solve;
type
  TByteSet = set of 0..7;
var
  BitCount: array[Byte] of Integer;
begin
end.
```

See also

Array types
Subrange types
Enumerated types
Case statement
Inc procedure
Dec procedure

"File type not allowed here"

See also Examples Compiler error messages

Description

File types are not allowed as value parameters and as the base type of a file type itself. They are also not allowed as function return types, and you cannot assign them - those errors will however produce a different error message.

Examples

{ In this example, the problem is that T is value parameter of type Text, which is a file type. Recall that whatever gets written to a value parameter has no effect on the caller's copy of the variable. Declaring a file as a value parameter therefore makes little sense. }

```
program Produce;

procedure WriteInteger(T: Text; I: Integer);
begin
  Writeln(T, I);
end;

begin
end.
```

{ Declaring the parameter as a var parameter solves the problem. }

```
program Solve;

procedure WriteInteger(var T: Text; I: Integer);
begin
  Writeln(T, I);
end;

begin
end.
```

See also
File types

"Low bound exceeds high bound"

Examples Compiler error messages

Description

This error message is given when either the low bound of a subrange type is greater than the high bound, or the low bound of a case label range is greater than the high bound.

Examples

{ In this example, the compiler gives an error rather than treating the ranges as empty. Most likely, the reversal of the bounds was not intentional. }

```
program Produce;
type
  SubrangeType = 1..0;              { Gets: Low bound exceeds high bound  }
begin
  case True of
  True..False:                      { Gets: Low bound exceeds high bound  }
    Writeln('Expected result');
  else
    Writeln('Unexpected result');
  end;
end.
```

{ Make sure you have specified the bounds in the correct order. }

```
program Solve;
type
  SubrangeType = 0..1;
begin
  case True of
  False..True:
    Writeln('Expected result');
  else
    Writeln('Unexpected result');
  end;
end.
```

"Program or unit recursively uses itself"

ExampleCompiler error messages

Description

This error message is issued if a **uses** clause ends up loading the same unit or program. This can happen when you specify the -P compiler directive (look for 8.3 names) due to file name truncation, if the names are identical in the first 8 characters.

Examples

{ The following unit, in the file Produce_.pas, produces an error when compiled with the use-short-file-names option }

```
unit Produce_Unit_1;
interface
uses Produce_Unit_2;
implementation
end.
```

"Procedure runs out of local address space"

Compiler error messages

Description

This error message is unused on the Intel architecture.

"Label '<name>' is not declared in current procedure"

Examples Compiler error messages

In contrast to Standard Pascal, Borland Pascal does not allow a **goto** to jump out of the current procedure. However, his construct is mainly useful for error handling, and Borland Pascal provides a more general and structured mechanism to deal with errors: exception handling.

Examples

{ The example above tries to halt computation by doing a non-local goto. }

```
program Produce;

label 99;

procedure MyProc;
begin
  { Something goes very wrong... }
  goto 99;
end;

begin
  MyProc;
  99:
    Writeln('Fatal error');
end.
```

{ In our solution, we used exception handling to stop the program. This has the advantage that we can also pass an error message. Another solution would be to use the standard procedures Halt or RunError. }

```
program Solve;

uses SysUtils;

procedure MyProc;
begin
  { Something goes very wrong... }
  raise Exception.Create('Fatal error');
end;

begin
  try
    MyProc;
  except
    on E: Exception do Writeln(E.Message);
  end;
end.
```

"Local procedure/function '<name>' assigned to procedure variable"

Examples Compiler error messages

Description

This error message is issued if you try to assign a local procedure to a procedure variable, or pass it as a procedural parameter. This is illegal, because the local procedure could then be called even if the enclosing procedure is not active. This situation would cause the program to crash if the local procedure tried to access any variables of the enclosing procedure.

Examples

{ The example tries to assign a local procedure to a procedure variable. This is illegal because it is unsafe at run time. }

```
program Produce;

var
  P: Procedure;

procedure Outer;

  procedure Local;
  begin
    Writeln('Local is executing');
  end;

begin
  P := Local;       { <-- Error message here }
end;

begin
  Outer;
  P;
end.
```

{ The solution is to move the local procedure out of the enclosing one. }

```
program Solve;

var
  P: Procedure;

procedure NonLocal;
begin
  Writeln('NonLocal is executing');
end;

procedure Outer;

begin
  P := NonLocal;
end;

begin
  Outer;
  P;
end.
```

"Missing ENDIF directive"

See also Examples Compiler error messages

Description

This error message is issued if the compiler does not find a corresponding **$ENDIF** directive after an **$IFDEF**, **$IFNDEF** or **$IFOPT** directive.

Examples

(* In this example, we left out the $ character in the **{$Endif}** directive, so the compiler mistook it for a comment. *)

```
program Produce;
{$AppType Console}
begin
{$IfOpt O+}
  Writeln('Compiled with optimizations');
{$Else}
  Writeln('Compiled without optimizations');
{Endif}
end.     { <-- Error message here }
```

{ The solution is to simply make sure all the conditional directives have a valid $ENDIF directive. }

```
program Solve;
{$AppType Console}
begin
{$IfOpt O+}
  Writeln('Compiled with optimizations');
{$Else}
  Writeln('Compiled without optimizations');
{$Endif}
end.
```

See also

Conditional directives

"Method identifier expected"

Examples Compiler error messages

Description

This error message will be issued in several different situations:

- Properties in an **automated** section must use methods for access, they cannot use fields in their **read** or **write** clauses.
- You tried to call a class method with the "ClassType.MethodName" syntax, but "MethodName" was not the name of a method.
- You tried calling an inherited method with the "**inherited** MethodName" syntax, but "MethodName" was not the name of a method.

Examples

{ The example tried to declare an automated property that accesses a field directly. The second error was caused by trying to get at a field of the base class - this is also not legal. }

```
program Produce;

type
  TMyBase = class
    Field: Integer;
  end;
  TMyDerived = class(TMyBase)
    Field: Integer;
    function Get: Integer;
  automated
    property Prop: Integer read Field;     { <-- Error message here }
  end;

function TMyDerived.Get: Integer;
begin
  Result := TMyBase.Field;                 { <-- Error message here }
end;

begin
end.
```

{ The first problem is fixed by accessing the field via a method. The second problem can be fixed by casting the Self pointer to the base class type, and acessing the field off of that. }

```
program Solve;

type
  TMyBase = class
    Field: Integer;
  end;
  TMyDerived = class(TMyBase)
    Field: Integer;
    function Get: Integer;
  automated
    property Prop: Integer read Get;
  end;

function TMyDerived.Get: Integer;
begin
  Result := TMyBase(Self).Field;
end;

begin
  Writeln( TMyDerived.Create.Prop );
end.
```

"Constant object cannot be passed as var parameter"

Examples Compiler error messages

Description

As variable parameters are intended to be modified by the called procedure or function, you can not pass a constant object to a variable parameter. If your intention is just to pass a big datastructure efficiently, and the called function should not modify it, you can use a const parameter instead.

Examples

{ In the example, function has a variable parameter, but we are passing a constant to it. }

```
program Produce;
{$AppType Console}

function Max(var A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
       Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );   { <-- Error message here }
end.
```

{ The solution is to declare the parameter as a constant parameter (we do not intend to modify it, after all). Alternatively, you can also modify the call so it does not pass constants. }

```
program Solve;
{$AppType Console}

function Max(const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
      Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );
end.
```

"FOR-Loop variable '<name>' cannot be passed as var parameter"

Examples Compiler error messages

Description

This warning is issued when you pass the control variable of a **for** loop as a variable parameter. The called procedure or function could modify the control variable and cause the for loop to execute incorrectly. Check whether the parameter should be better declared as value or const parameter. If that is possible, it will improve the efficiency of both the called procedure and the **for** loop.

Examples

{ The example passes the control variable I as a variable parameter to procedure MyProc, which causes a warning. }

```
program Produce;
{$WARNINGS ON}

procedure MyProc(var I: Integer);
begin
  Writeln(I);
end;

var
  I: Integer;
begin
  for I := 1 to 10 do
    MyProc(I);            { <-- Warning message here }
end.
```

{ The solution in this case was to change the parameter declaration to a constant parameter. Alternatively, we can also assign the control variable to another, auxiliary variable, and pass that one. }

```
program Solve;
{$WARNINGS ON}

procedure MyProc(const I: Integer);
begin
  Writeln(I);
end;

var
  I: Integer;
begin
  for I := 1 to 10 do
    MyProc(I);
end.
```

"BREAK or CONTINUE outside of loop"

Examples Compiler error messages

Description

The compiler has found a Break or Continue statement which is not contained inside a **while** or **repeat** loop. These two constructs are only legal in loops.

Examples

{ The example above shows how a continue statement could seem to be included in the body of a looping construct but, due to the compound-statement nature of Pascal, it really is not. }

```
program Produce;

  procedure Error;
  var
    I: Integer;
  begin
    I := 0;
    while I < 100 do
      Inc(I);
      if Odd(I) then
      begin
        Inc(I);
        Continue;
      end;
  end;

begin
end.
```

{ Often times it is a simple matter to create compound statement out of the looping construct to ensure that your CONTINUE or BREAK statements are included. }

```
program Solve;

  procedure Error;
  var
    I: Integer;
  begin
    I := 0;
    while I < 100 do
    begin
      Inc(I);
      if Odd(I) then
      begin
        Inc(I);
        Continue;
      end;
    end;
  end;

begin
end.
```

"Division by zero"

Compiler error messages

Description

The compiler has detected a constant division by zero in your program.
Check your constant expressions and respecify them so that a division by zero error will not occur.

"Type of expression must be BOOLEAN"

Examples Compiler error messages

Description

This error message is output when an expression serves as a condition and must therefore be of Boolean type. This is the case for the controlling expression of the if, while and repeat statements, and for the expression that controls a conditional breakpoint.

Examples

{ Here, a C++ programmer just used a pointer variable as the condition of an if statement. }

```
program Produce;
var
  P: Pointer;
begin
  if P then
    Writeln('P <> nil');
end.
```

{ In Pascal, you need to be more explicit in this case. }

```
program Solve;
var
  P: Pointer;
begin
  if P <> nil then
    Writeln('P <> nil');
end.
```

"Overflow in conversion or arithmetic operation"

Compiler error messages

Description

The compiler has detected an overflow in an arithmetic expression: the result of the expression is too large to be represented in 32 bits.
Check your computations to ensure that the value can be represented by the computer hardware.

"Data type too large: exceeds 2 GB"

Examples Compiler error messages

Description

You have specified a data type which is too large for the compiler to represent. You must decrease the size of the description of the type.

Examples

{ It is easily apparant to see why these declarations will elicit error messages. }

```
program Produce;

type
  EnormousArray = array[0..MaxLongint] of Longint;
  BigRecord = record
    Points: array[1..10000] of Extended;
  end;

var
  Data: array[0..500000] of BigRecord;

begin
end.
```

{ The easy solution to avoid this error message is to make sure that the size of your data types remain under 2Gb in size. A more complicated method would involve the restructuring of your data, as has been begun with the BigRecord declaration. }

```
program Solve;
type
  EnormousArray = array[0..MaxLongint div 8] of Longint;
  DataPoints = ^DataPointDesc;
  DataPointDesc = array[1..10000] of Extended;
  BigRecord = record
    Points: DataPoints;
  end;

var
  Data: array[0..500000] of BigRecord;

begin
end.
```

"Integer constant too large"

Examples Compiler error messages

Description

You have specified an integer constant that requires more than 32 bits to represent.

Examples

{ Both constants in the following example are too large to represent in 32 bits, thus the compiler will output an error.
}

```
program Produce;

const
  VeryDecimal = 123456789;
  VeryBigHex = $123456789;

begin
end.
```

{ Check the constants that you have specified and ensure that they are representable in 32 bits. }

```
program Solve;

const
  VeryDecimal = 12345678;
  VeryBigHex = $12345678;

begin
end.
```

"16-Bit fixup encountered in object file '<Filename>'"

Compiler error messages

Description

A 16-bit fixup has been found in one of the object modules linked to your program with the **$L** compiler directive.
The compiler only supports 32 bit fixups in linked object modules.
Make sure that the linked object module is a 32 bit object module.

"Inline assembler syntax error"

Examples Compiler error messages

Description

You have entered an expression which the inline assembler is unable to interpret as a valid assembly instruction. Examine the offending inline assembly statement and ensure that it conforms to the proper syntax.

Examples

```
program Produce;

  procedure Assembly;
  asm
    adx  eax, 151
  end;

begin
end.

program Solve;

  procedure Assembly;
  asm
    add  eax, 151
  end;

begin
end.
```

"Inline assembler stack overflow"

Compiler error messages

Description

Your inline assembler code has exceeded the capacity of the inline assembler.
Contact Borland International if you encounter this error.

"Operand size mismatch"

See also Examples Compiler error messages

Description

The size required by the instruction operand does not match the size given.

Examples

{ In the following sample, the compiler will complain because the 'offset' operator produces a 'dword', but the operator is expecting a 'byte'. }

```
program Produce;

var
  V: Integer;

  procedure Assembly;
  asm
    db offset V
  end;

begin
end.
```

{ The solution, for this example, is to change the operator to receive a 'dword'. In the general case you will need to closely examine your code and ensure the the operator and operand sizes match. }

```
program Solve;

var
  V: Integer;

  procedure Assembly;
  asm
    dd offset v
  end;

begin
end.
```

See also
Inline assembler

"Memory reference expected"

Compiler error messages

Description

The inline assembler has expected to find a memory reference expression but did not find one.
Ensure that the offending statement is indeed a memory reference.

"Constant expected"

Examples Compiler error messages

Description

The inline assembler was expecting to find a constant but did not find one.
Many of the inline assembler expressions require constants to assemble correctly. Change the offending statement to have a assemble-time constant.

Examples

{ The inline assembler is not capable of performing a MOD operation on a Pascal variable, thus the following code will cause an error. }

```
program Produce;

  procedure Assembly(X: Integer);
  asm
    mov   ax, X MOD 10
  end;

begin
end.
```

"Type expected"

Compiler error messages

Description

Contact Borland International if you receive this error.

"Type of expression must be INTEGER"

Examples Compiler error messages

Description

This error message is only given when the constant expression that specifies the number of characters in a short string type is not of type integer.

Examples

{ The example tries to specify the number of elements in a string as dependent on the maximum element of type Color - unfortunately, the element count is of type Color, which is illegal. }

```
program Produce;
type
  Color = (red,green,blue);
var
  S3: string[Succ(High(Color))];
begin
end.

program Solve;
type
  Color = (red,green,blue);
var
  S3: string[Ord(High(Color))+1];
begin
end.
```

"Cannot add or subtract relocatable symbols"

ExampleCompiler error messages

Description

The inline assembler is not able to add or substract memory address which may be changed by the linker.
Make sure you don't try to add or subtract relocatable addresses from within your inline assembler statements.

Example

{ Global variables fall into the class of items which produce relocatable addresses, and the inline assembler is unable to add or subtract these. }

```
program Produce;

var
  A: array[1..10] of Integer;
  EndOfA: Integer;

  procedure Relocatable;
  begin
  end;

  procedure Assembly;
  asm
    mov eax, A + EndOfA
  end;

begin
end.
```

"Invalid register combination"

Examples Compiler error messages

Description

You have specified an illegal combination of registers in a inline assembler statement. Please refer to an assembly language guide for more information on addressing modes allowed on the Intel 80x86 family.

Examples

{ The right operand specified in this mov instruction is illegal. }

```
program Produce;

  procedure AssemblerExample;
  asm
    mov eax, [ecx + esp * 4]
  end;

begin
end.
```

{ The addressing mode specified by the right operand of this mov instruction is allowed. }

```
program Solve;

  procedure AssemblerExample;
  asm
    mov eax, [ecx + ebx * 4]
  end;

begin
end.
```

"Numeric overflow"

Examples Compiler error messages

Description

The inline assembler has detected a numeric overflow in one of your expressions.

Examples

{ Specifying a number which requires more than 32bits to represent will elicit this error. }

```
program Produce;

  procedure AssemblerExample;
  asm
    mov eax, $0fffffffffffffffffffff
  end;

begin
end.
```

{ Make sure that your numbers all fit in 32bits. }

```
program Solve;

  procedure AssemblerExample;
  asm
    mov al, $0ff
  end;

begin
end.
```

"String constant too long"

Examples Compiler error messages

Description

The inline assembler has not found the end of the string that you specified. The most likely cause is a misplaced closing quote.

Examples

{ The inline assembler is unable to find the end of the string, before the end of the line, so it reports that the string is too long. }

```
program Produce;

  procedure AssemblerExample;
  asm
    db 'Hello world.  I am an inline assembler statement
  end;

begin
end.
```

{ Adding the closing quote will vanquish this error. }

```
program Solve;

  procedure AssemblerExample;
  asm
    db 'Hello world.  I am an inline assembler statement'
  end;

begin
end.
```

"Error in numeric constant"

Examples Compiler error messages

Description

The inline assembler has found an error in the numeric constant you entered.

Examples

{ In the following example, the inline assembler was expecting to parse a hexadecimal constant, but it found an erroneous character. }

```
program Produce;

  procedure AssemblerExample;
  asm
    mov al, $z0f0
  end;

begin
end.
```

{ Make sure that the numeric constants you enter conform to the type that the inline assembler is expecting to parse. }

```
program Solve;

  procedure AssemblerExample;
  asm
    mov al, $f0
  end;

begin
end.
```

"Invalid combination of opcode and operands"

Examples Compiler error messages

Description

You have specified an inline assembler statement which is not correct.

Examples

{ The inline assembler is not capable of storing the result of $f0 * 16 into the 'al' register -- it simply won't fit. }

```
program Produce;

  procedure AssemblerExample;
  asm
    mov al, $0f0 * 16
  end;

begin
end.
```

{ Make sure that the type of both operands are compatible. }

```
program Solve;
  procedure AssemblerExample;
  asm
    mov al, $0f * 16
  end;

begin
end.
```

"486/487 instructions not enabled"

Compiler error messages

Description

You should not receive this inline assembler error as 486 instructions are always enabled.

"Division by zero"

Examples Compiler error messages

Description

The inline assembler has encountered an expression which results in a division by zero.

Examples

{ If you are using program constants instead of constant literals, this error might not be quite so obvious. }

```
program Produce;

  procedure AssemblerExample;
  asm
    dw 1000 / 0
  end;

begin
end.
```

{ The solution, as when programming in high level langauges, is to make sure that you don't divide by zero. }

```
program Solve;

  procedure AssemblerExample;
  asm
    dw 1000 / 10
  end;

begin
end.
```

"Structure field identifier expected"

Examples Compiler error messages

Description

The inline assembler recognized an identifier on the right side of a '.', but it was not a field of the record found on the left side of the '.'. One common, yet difficult to realize, error of this sort is to use a record with a field called 'ch' -- the inline assembler will always interpret 'ch' to be a register name.

Examples

{ In this example, the inline assembler has recognized that 'y' is a valid identifier, but it has not found 'y' to be a member of the type of 'd'. }

```
program Produce;

type
  Data = record
    X: Integer;
  end;

  procedure AssemblerExample(D: Data; Y: Char);
  asm
    mov  eax, D.Y
  end;

begin
end.
```

{ By specifying the proper variable name, the error will go away. }

```
program Solve;

type
  Data = record
    X: Integer;
  end;

  procedure AssemblerExample(D: Data; Y: Char);
  asm
    mov  eax, D.X
  end;

begin
end.
```

"LOOP/JCXZ distance out of range"

Compiler error messages

Description

You have specified a LOOP or JCXZ destination in inline assembler code that is out of range. The distance must be in the range -128..127.
If you encounter this error, replace LOOP or JCXZ with an equivalent two-instruction sequence.

"Statement expected, but expression of type '<type>' found"

Examples Compiler error messages

Description

The compiler was expecting to find a statement, but instead it found an expression of the specified type.

Examples

{ In this example, the compiler is expecting to find a statement, such as an IF, WHILE, REPEAT, but instead it found the expression (3+4). }

```
program Produce;
var
  A: Integer;
begin
  (3 + 4);
end.
```

{ The solution here was to assign the result of the expression (3+4) to the variable 'a'. Another solution would have been to remove the offending expression from the source code. The choice depends on the situation. }

```
program Produce;
var
  A: Integer;
begin
  A := (3 + 4);
end.
```

"Procedure or function name expected"

Examples Compiler error messages

Description

You have specified an identifier which does not represent a procedure or function in an **exports** clause.

Examples

```
{ It is not possible to export variables from a Delphi library, even though the variable is of 'procedure' type. }
library Produce;

var
  Y: procedure;

exports Y;
begin
end.

{ Always be sure that all the identifiers listed in an EXPORTS clause truly represent procedures. }
program Solve;

  procedure ExportMe;
  begin
  end;

exports ExportMe;
begin
end.
```

"PROCEDURE or FUNCTION expected"

Examples Compiler error messages

Description

This error message is produced by two different constructs, but in both cases the compiler is expecting to find the keyword **procedure** or the keyword **function**.

Examples

{ In both cases above, the word 'procedure' should follow the keyword 'class'. }

```
program Produce;

type
  Base = class
    class AProcedure; { case 1 }
  end;

  class Base.AProcedure; { case 2 }
  begin
  end;

begin
end.
```

{ As can be seen, adding the keyword 'procedure' removes the error from this program. }

```
program Solve;

type
  Base = class
    class procedure AProcedure;
  end;

  class procedure Base.AProcedure;
  begin
  end;

begin
end.
```

"Instance variable '<name>' inaccessible here"

Examples Compiler error messages

Description

You are attempting to reference an instance variable from within a class method.

Examples

{ Class methods do not have an instance pointer, so they cannot access any methods or instance data of the class. }

```
program Produce;

type
  Base = class
    Title: string;
    class procedure Init;
  end;

  class procedure Base.Init;
  begin
    Self.Title := 'Does not work';
    Title := 'Does not work';
  end;

begin
end.
```

{ The only solution to this error is to not access any member data or methods from within a class method. }

```
program Solve;

type
  Base = class
    Title: string;
    class procedure Init;
  end;

  class procedure Base.Init;
  begin
  end;

begin
end.
```

"EXCEPT or FINALLY expected"

Examples Compiler error messages

Description

Every **try** block must contain either an exception-handling part **(except)** or a cleanup code part **(finally)**.

Examples

{ In the code above, the 'except' or 'finally' clause of the exception handling code is missing, so the compiler will issue an error. }

```
program Produce;

begin
  try
  end;
end.
```

{ By adding the missing clause, the compiler will be able to complete the compilation of the code. In this case, the 'except' clause will allow the program to finish. }

```
program Solve;

begin
  try
  except
  end;
end.
```

"Cannot BREAK, CONTINUE or EXIT out of a FINALLY clause"

Examples Compiler error messages

Description

Because a **finally** clause may be entered and exited through Delphi's exception handling mechanism or through normal program control, the explicit control flow of your program may not be followed. When the finally is entered through the exception handling mechanism, it is not possible to exit the clause with Break, Continue, or Exit -- when the finally clause is being executed by the exception handling system, control must return to the exception handling system.

Examples

{ The following program attempts to exit the finally clause with a break statement. It is not legal to exit a FINALLY clause in this manner. }

```
program Produce;

  procedure A0;
  begin
    try
      (* try something that might fail *)
    finally
      Break;
    end;
  end;

begin
end.
```

{ The only solution to this error is to restructure your code so that the offending statement does not appear in the FINALLY clause. }

```
program Solve;

  procedure A0;
  begin
    try
      (* try something that might fail *)
    finally
    end;
  end;

begin
end.
```

"'GOTO <label>' leads into or out of TRY statement"

ExampleCompiler error messages

Description

The **goto** statement cannot jump into or out of an exception handling statement.
The ideal solution to this problem is to avoid using **goto** statements altogether, however, if that is not possible you will have to perform more detailed analysis of the program to determine the correct course of action.

Example

{ Both GOTO statements in the following code are incorrect. It is not possible to jump into, or out of, exception handling blocks. }

```
program Produce;

label 1, 2;

begin
  goto 1;
  try
1:
  except
    goto 2;
  end;
2:
end.
```

"<clause1> clause expected, but <clause2> found"

Examples Compiler error messages

Description

The compiler was, due to the Pascal syntax, expecting to find a <clause1> in your program, but instead found <clause2>.

Examples

{ The first declaration of a property must specify a read and write clause, and since both are missing on the 'Ch' property, an error will result when compiling. In the case of properties, the original intention might have been to hoist a property defined in a base class to another visibility level -- for example, from public to private. In this case, the most probable cause of the error is that the property name was not found in the base class. Make sure that you have spelled the property name correctly and that it is actually contained in one of the parent classes. }

```
program Produce;
type
  CharDesc = class
    vch: Char;
    property Ch: Char;
  end;
end.
```

{ The solution is to ensure that all the proper clauses are specified, where required. }

```
program Produce;

type
  CharDesc = class
    vch: Char;
    property Ch: Char read vch write vch;
  end;
end.
```

"Cannot assign to a read-only property"

Examples Compiler error messages

Description

The property to which you are attempting to assign a value did not specify a **write** clause, thereby causing it to be a read-only property.

Examples

{ If a property does not specify a 'write' clause, it effectively becomes a read-only property; it is not possible to assign a value to a property which is read-only, thus the compiler outputs an error on the assignment to 'c.Title'. }

```
program Produce;

type
  Base = class
    S: string;
    property Title: string read S;
  end;

var
  C: Base;

procedure DiddleTitle
begin
  if C.Title = '' then
    C.Title := 'Super Galactic Invaders with Turbo Gungla Sticks';
    { perform other work on the C.Title }
end;

begin
end.
```

{ One solution, if you have source code, is to provide a write clause for the read-only property -- of course, this could dramatically alter the semantics of the base class and should not be taken lightly. Another alternative would be to introduce an intermediate variable which would contain the value of the read-only property -- it is this second alternative which is shown in the code that follows. }

```
program Solve;

type
  Base = class
    S: string;
    property Title: string read S;
  end;

var
  C: Base;

procedure DiddleTitle;
var
  Title: string;
begin
  Title := C.Title;
  if Title = '' then
    Title := 'Super Galactic Invaders with Turbo Gungla Sticks';
    { perform other work on Title }
end;

begin
end.
```

"Cannot read a write-only property"

Examples Compiler error messages

Description

The property from which you are attempting to read a value did not specify a **read** clause, thereby causing it to be a write-only property.

Examples

{ Since C.Password has not specified a read clause, it is not possible to read its value. }

```
program Produce;

type
  Base = class
    S: string;
    property Password: string write S;
  end;

var
  C: Base;
  S: string;

begin
  S := C.Password;
end.
```

{ One easy solution to this problem, if you have source code, would be to add a read clause to the write-only property. But, adding a read clause is not always desirable and could lead to holes in a security system -- consider, for example, a write-only property called 'Password', as in this example: you certainly wouldn't want to casually allow programs using this class to read the stored password. If a property was created as write-only, there is probably a good reason for it and you should reexamine why you need to read this property. }

```
program Solve;

type
  Base = class
    S: string;
    property Password: string read S write S;
  end;

var
  C: Base;
  S: string;

begin
  S := C.Password;
end.
```

"Class already has a default property"

See also Examples Compiler error messages

Description

You have tried to assign a default property to a class which already has defined a default property.

Examples

{ The Access property in the code above attempts to become the default property of the class, but Data has already been specified as the default. There can be only one default property in a class. }

```
program Produce;

type
  Base = class
    function GetV(I: Integer): Char;
    procedure SetV(I: Integer; const X: Char);
    property Data[I: Integer]: Char read GetV write SetV; default;
    property Access[I: Integer]: Char read GetV write SetV; default;
  end;

function Base.GetV(I: Integer): Char;
begin
  GetV := 'A';
end;

procedure Base.SetV(I: Integer; const X: Char);
begin
end;

begin
end.
```

{ The solution is to remove the incorrect default property specifications from the program source. }

```
program Solve;

type
  Base = class
    function GetV(I: Integer): Char;
    procedure SetV(I: Integer; const X: Char);
    property Data[I: Integer]: Char read GetV write SetV; default;
  end;

function Base.GetV(I: Integer): Char;
begin
  GetV := 'A';
end;

procedure Base.SetV(I: Integer; const X: Char);
begin
end;

begin
end.
```

See also

Default property

Properties

Array Properties

"Operator not applicable to this operand type"

Examples Compiler error messages

Description

This error message is given whenever an operator cannot be applied to the operands it was given - for instance if a boolean operator is applied to a pointer.

Examples

{ Here a C++ programmer was unclear about operator precedence in Pascal; P is not a Boolean expression, and the comparison needs to be parenthesized. }

```
program Produce;
var
  P: ^Integer;
begin
  if P and P^ > 0 then
    Writeln('P points to a number greater 0');
end.
```

{ If we explicitly compare P to nil and use parentheses, the compiler is happy. }

```
program Solve;
var
  P: ^Integer;
begin
  if (P <> nil) and (P^ > 0) then
    Writeln('P points to a number greater 0');
end.
```

"Default property must be an array property"

Examples Compiler error messages

Description

The default property which you have specified for the class is not an array property. Default properties are required to be array properties.

Examples

{ When specifying a default property, you must make sure that it conforms to the array property syntax. The Data property in the following code specifies a Char type rather than an array. }

```
program Produce;

type
  Base = class
    function GetV: Char;
    procedure SetV(X: Char);
    property Data: Char read GetV write SetV; default;
  end;

function Base.GetV: Char;
begin
  GetV := 'A';
end;

procedure Base.SetV(X: Char);
begin
end;

begin
end.
```

{ By changing the specifciation of the offending property to an array, or by removing the 'default' directive, you can remove this error. }

```
program Solve;

type
  Base = class
    function GetV(I: Integer): Char;
    procedure SetV(I: Integer; const X: Char);
    property Data[I: Integer]: Char read GetV write SetV; default;
  end;

function Base.GetV(I: Integer): Char;
begin
  GetV := 'A';
end;

procedure Base.SetV(I: Integer; const X: Char);
begin
end;

begin
end.
```

"TYPEINFO standard function expects a type identifier"

Examples Compiler error messages

Description

You have attempted to obtain type information for an identifier which does not represent a type.

Examples

{ The TypeInfo standard procedure requires a type identifer as it's parameter. In the code that follows, 'NotType' does not represent a type identifier. }

```
program Produce;

var
  P: Pointer;

procedure NotType;
begin
end;

begin
  P := TypeInfo(NotType);
end.
```

{ By ensuring that the parameter used for TypeInfo is a type identifier, you will avoid this error. }

```
program Solve;

type
  Base = class
  end;

var
  P: Pointer;

begin
  P := TypeInfo(Base);
end.
```

"Type '<name>' has no type info"

Examples Compiler error messages

Description

You have applied the TypeInfo standard procedure to a type identifier which does not have any run-time type information associated with it.

Examples

{ Record types do not generate type information, so this use of TypeInfo is illegal. }

```
program Produce;

type
  Data = record
  end;

var
  V: Pointer;

begin
  V := TypeInfo(Data);
end.
```

{ A class does generate RTTI, so the use of TypeInfo here is perfectly legal. }

```
program Solve;

type
  Base = class
  end;

var
  V: Pointer;

begin
  V := TypeInfo(Base);
end.
```

"FOR or WHILE loop executes zero times - deleted"

Examples Compiler error messages

Description

The compiler has determined that the specified looping structure will not ever execute, so as an optimization it will remove it.

Examples

{ The compiler determines that 'FALSE AND (i < 100)' always evaluates to FALSE, and then easily determines that the loop will not be executed. }

```
program Produce;
{$HINTS ON}

var
  I: Integer;

begin
  I := 0;
  while False and (I < 100) do
    Inc(I);
end.
```

{ The solution to this hint is to check the boolean expression used to control while statements is not always FALSE. In the for loops you should make sure that (upper bound - lower bound) >= 1. }

```
program Solve;
{$HINTS ON}

var
  I: Integer;

begin
  I := 0;
  while I < 100 do
    Inc(I);
end.
```

"No definition for abstract method '<name>' allowed"

Examples Compiler error messages

Description

You have declared <name> to be abstract, but the compiler has found a definition for the method in the source file. It is illegal to provide a definition for an abstract declaration.

Examples

{ Abstract methods cannot be defined. An error will appear at the point of Base.Foundation when you compile this program. }

```
program Produce;

type
  Base = class
    procedure Foundation; virtual; abstract;
  end;

procedure Base.Foundation;
begin
end;

begin
end.
```

{ Two steps are required to solve this error. First, you must remove the definition of the abstract procedure which is declared in the base class. Second, you must extend the base class, declare the abstract procedure as an 'override' in the extension, and then provide a definition for the newly declared procedure. }

```
program Solve;

type
  Base = class
    procedure Foundation; virtual; abstract;
  end;

  Derived = class(Base)
    procedure Foundation; override;
  end;

procedure Derived.Foundation;
begin
end;

begin
end.
```

"Method '<name>' not found in base class"

Examples Compiler error messages

Description

You have applied the 'override' directive to a method, but the compiler is unable to find a procedure of the same name in the base class.

Examples

{ A common cause of this error is a simple typographical error in your source code. Make sure that the name used as the 'override' procedure is spelled the same as it is in the base class. In other situations, the base class will not provide the desired procedure: it is those situations which will require much deeper analysis to determine how to solve the problem. }

```
program Produce;

type
  Base = class
    procedure Title; virtual;
  end;

  Derived = class(Base)
    procedure Titl; override;
  end;

procedure Base.Title;
begin
end;

procedure Derived.Titl;
begin
end;

begin
end.
```

{ The solution in this example is to simple correct the spelling of the procedure name in Derived. }

```
program Solve;

type
  Base = class
    procedure Title; virtual;
  end;

  Derived = class(Base)
    procedure Title; override;
  end;

procedure Base.Title;
begin
end;

procedure Derived.Title;
begin
end;

begin
end.
```

"Invalid message parameter list"

Examples Compiler error messages

Description

A message-handling method can take only one, var, parameter; it's type is not checked.

Examples

{ The obvious error in the first case is that the parameter is not VAR. The error in the second case is that more than one parameter is declared. }

```
program Produce;

type
  Base = class
    procedure Msg1(X: Integer); message 151;
    procedure Msg2(var X, Y: Integer); message 152;
  end;

procedure Base.Msg1(X: Integer);
begin
end;

procedure Base.Msg2(var X, Y: Integer);
begin
end;

begin
end.
```

{ The solution in both cases was to only specify one, var, parameter in the message method declaration. }

```
program Solve;

type
  Base = class
    procedure Msg1(var X: Integer); message 151;
    procedure Msg2(var Y: Integer); message 152;
  end;

procedure Base.Msg1(var X: Integer);
begin
end;

procedure Base.Msg2(var Y: Integer);
begin
end;

begin
end.
```

"Illegal message method index"

See also Examples Compiler error messages

Description

You have specified value for your message index which <= 0.

Examples

{ The specification of -151 as the message index is illegal in the above example. }

```
program Produce;

type
  Base = class
    procedure Dynamo(var X: Integer); message -151;
  end;

procedure Base.Dynamo(var X: Integer);
begin
end;

begin
end.
```

{ Always make sure that your message index values are >= 1. }

```
program Solve;

type
  Base = class
    procedure Dynamo(var X: Integer); message 151;
  end;

procedure Base.Dynamo(var X: Integer);
begin
end;

begin
end.
```

See also

Message methods

Virtual methods

Dynamic methods

"Duplicate dynamic method index"

Examples Compiler error messages

Description

You have specified an index for a dynamic methodwhich is already used by another dynamic method.

Examples

{ The declaration of 'Second' attempts to reuse the same message index which is used by 'First'; this is illegal. }

```
program Produce;

type
  Base = class
    procedure First(var X: Integer); message 151;
    procedure Second(var X: Integer); message 151;
  end;

procedure Base.First(var X: Integer);
begin
end;

procedure Base.Second(var X: Integer);
begin
end;

begin
end.
```

{ There are two straightforward solutions to this problem. First, if you really do not need to use the same message value, you can simply change the message number to be unique. Alternatively, you could derive a new class from the base and override the behavior of the message handler declared in the base class. Both options are shown in the above example. }

```
program Solve;

type
  Base = class
    procedure First(var X: Integer); message 151;
    procedure Second(var X: Integer); message 152; { change to unique
  index }
  end;

  Derived = class(Base)
    procedure First(var X: Integer); override; { override base class
  behavior }
  end;

procedure Base.First(var X: Integer);
begin
end;

procedure Base.Second(var X: Integer);
begin
end;

procedure Derived.First(var X: Integer);
begin
end;

begin
end.
```

"Bad file format '<name>'"

Compiler error messages

Description

The compiler state file has become corrupted. It is not possible to reload the previous compiler state.

"Array type required"

Examples Compiler error messages

Description

This error message is given if you either index into an operand that is not an array, or if you pass an argument that is not an array to an open array parameter.

Examples

{ We try to apply an index to a pointer to integer - that would be legal in C, but is not in Pascal. }

```
program Produce;
var
  P: ^Integer;
  I: Integer;
begin
  Writeln(P[I]);
end.
```

{ In Pascal, we must tell the compiler that we intend P to point to an array of integers. }

```
program Solve;
type
  TIntArray = array[0..MaxInt div sizeof(Integer)-1] of Integer;
var
  P: ^TIntArray;
  I: Integer;
begin
  Writeln(P^[I]); { P[I] would also be legal in Delphi 2.0 }
end.
```

"Inaccessible value"

Compiler error messages

Description

You have tried to view a value that is not accessible from within the integrated debugger. Certain types of values, such as a zero-length Variant-type string, cannot be viewed within the debugger.

"Destination cannot be assigned to"

Compiler error messages

Description

The integrated debugger has determined that your assignment is not valid in the current context.

"Expression has no value"

Compiler error messages

Description

You have attempted to assign the result of an expression, which did not produce a value, to a variable.

"Destination is inaccessible"

Compiler error messages

Description

The address to which you are attempting to put a value is inaccessible from within the integrated debugger.

"Expression is not a procedure"

Compiler error messages

Description

You have attempted to use a symbol from your program as if it were a procedure, but it is not.

"No source line for this procedure"

Compiler error messages

Description

The integrated debugger is unable to find a source line for the procedure you requested.
If you have the source, recompile the unit containing the desired procedure with debugging information turned on.
If you don't have the source, you will not be able to view this procedure.

"Re-raising an exception only allowed in exception handler"

Examples Compiler error messages

Description

You have used the syntax of the raise statement which is used to reraise an exception, but the compiler has determined that this reraise has occurred outside of an exception-handling block. A limitation of the current exception handling mechanism disallows reraising exceptions from nested exception handlers.

Examples

{ There are several reasons why this error might occur. First, you might have specified a raise with no exception constructor outside of an exception handler. Secondly, you might be attempting to reraise an exception in the try block of an exception handler. Thirdly, you might be attempting to reraise the exception in an exception handler nested in another exception handler. }

```
program Produce;

procedure RaiseException;
begin
  raise;             { case 1 }
  try
    raise;           { case 2 }
  except
    try
      raise;         { case 3 }
    except
    end;
    raise;
  end;
end;

begin
end.
```

{ One solution to this error is to explicitly raise a new exception; this is probably the intention in situations like 'case 1' and 'case 2'. For the situation of 'case 3', you will have to examine your code to determine a suitable workaround which will provide the desired results. }

```
program Solve;
uses SysUtils;

procedure RaiseException;
begin
  raise Exception.Create('case 1');
  try
    raise Exception.Create('case 2');
  except
    try
      raise Exception.Create('case 3');
    except
    end;
    raise;
  end;
end;

begin
end.
```

"Default values must be of ordinal, pointer or small set type"

Examples Compiler error messages

Description

You have declared a property containing a **default** clause, but the property type is incompatible with default values.

Examples

{ The program above creates a property and attempts to assign a default value to it, but since the type of the property does not allow default values, an error is output. }

```
program Produce;

type
  VisualGuage = class
    Pos: Single;
    property Position: Single read Pos write Pos default 0.0;
  end;

begin
end.
```

{ When this error is encountered, there are two easy solutions: the first is to remove the default value definition, and the second is to change the type of the property to one which allows a default value. Your program, however, may not be as simple to fix; consider when you have a set property which is too large -- it is this case which will require you to carefully examine your program to determine the best solution to this problem. }

```
program Solve;

type
  VisualGuage = class
    Pos: Integer;
    property Position: Integer read Pos write Pos default 0;
  end;

begin
end.
```

"Property '<name>' does not exist in base class"

Examples Compiler error messages

Description

The compiler believes you are attempting to hoist a property to a different visibility level in a derived class, but the specified property does not exist in the base class.

Examples

{ There are two basic causes of this error. The first is the specification of a new property without specifying a type; this usually is not supposed to be a movement to a new visibility level. The second is the specification of a property which should exist in the base class, but is not found by the compiler; the most incarnation of this is a typo. In the second form, the compiler will also output errors that a read or write clause was expected. of a proper }

```
program Produce;

type
  Base = class
  private
    A: Integer;
    property BaseProp: Integer read A write A;
  end;

  Derived = class(Base)
    Ch: Char;
    property Alpha read Ch write Ch; { case 1 }
    property BesaProp; { case 2 }
  end;

begin
end.
```

{ The solution for the first case is to supply the type of the property. The solution for the second case is to check the spelling of the property name. }

```
program Solve;

type
  Base = class
  private
    A: Integer;
    property BaseProp: Integer read A write A;
  end;

  Derived = class(Base)
    Ch: Char;
  public
    property Alpha: Char read Ch write Ch; { case 1 }
    property BaseProp; { case 2 }
  end;

begin
end.
```

"Dynamic method or message handler not allowed here"

Examples Compiler error messages

Description

Dynamic and message methods cannot be used as accessor functions for properties.

Examples

{ Both 'Velocity' and 'Value' above are in error since they both have illegal accessor functions assigned to them. }

```
program Produce;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer); dynamic;
    function GetV: Integer; message;
    property Velocity: Integer read GetV write V;
    property Value: Integer read V write SetV;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

{ The solution taken in this is example was to remove the offending compiler directives from the procedure declarations; this may not be the right solution for you. You may have to closely examine the logic of your program to determine how best to provide accessor functions for your properties. }

```
program Solve;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
    property Velocity: Integer read GetV write V;
    property Value: Integer read V write SetV;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := v;
end;

begin
end.
```

"Pointer type required"

Examples Compiler error messages

Description

This error message is given when you apply the dereferencing operator '^' to an operand that is not a pointer, and, as a very special case, when the second operand in a 'Raise <exception> at <address>' statement is not a pointer.

Examples

{ Even though class types are implemented internally as pointers to the actual information, it is illegal to apply the dereferencing operator to class types at the source level. It is also not necessary - the compiler will dereference automatically whenever it is appropriate. }

```
program Produce;
var
  C: TObject;
begin
  C^.Destroy;
end.
```

{ Simply leave off the dereferencing operator - the compiler will do the right thing automatically. }

```
program Solve;
var
  C: TObject;
begin
  C.Destroy;
end.
```

"Class does not have a default property"

Examples Compiler error messages

Description

You have used a class instance variable in an array expression, but the class type has not declared a default array property.

Examples

{ The example above elicits an error because 'Base' does not declare an array property, and 'b' is not an array itself. }

```
program Produce;

type
  Base = class
  end;

var
  B: Base;

procedure P;
var
  Ch: Char;
begin
  Ch := B[1];
end;

begin
end.
```

{ When you have declared a default property for a class, you can use the class instance variable in array expression, as if the class instance variable itself were actually an array. Alternatively, you can use the name of the property as the actual array accessor. Note: if you have hints turned on, you will receive two warnings about the value assigned to 'ch' never being used. }

```
program Solve;

type
  Base = class
    function GetChar(I: Integer): Char;
    property data[I: Integer]: Char read GetChar; default;
  end;

var
  B: Base;

function Base.GetChar(I: Integer): Char;
begin
  GetChar := 'A';
end;

procedure P;
var
  Ch: Char;
begin
  Ch := B[1];
  Ch := B.Data[1];
end;

begin
end.
```

"Bad argument type in variable type array constructor"

Examples Compiler error messages

Description

You are attempting to construct an array using a type which is not allowed in variable arrays.

Examples

{ Both calls to Examiner will fail because enumerations and records are not supported in array constructors. }

```
program Produce;

type
  Fruit = (apple, orange, pear);
  Data = record
    X: Integer;
    Ch: Char;
  end;

var
  F: Fruit;
  D: Data;

procedure Examiner(V: array of TVarRec);
begin
end;

begin
  Examiner([D]);
  Examiner([F]);
end.
```

{ Many data types, like those in the example above, are allowed in array constructors. }

```
program Solve;

var
  I: Integer;
  R: Real;
  V: Variant;

procedure Examiner(V: array of TVarRec);
begin
end;

begin
  I := 0; R := 0; V := 0;
  Examiner([I, R, V]);
end.
```

"Could not load RLINK32.DLL"

Compiler error messages

Description

RLINK32.DLL could not be found. Please ensure that it is on the path.

"Wrong or corrupted version of RLINK32.DLL"

Compiler error messages

Description

The internal consistency check performed on the RLINK32.DLL file has failed.
You should check to insure that you have no conflicts on your PATH which could be causing an older RLINK32 to load. If you are sure that you have a proper RLINK32.DLL, and you still receive this message, please contact Borland International.

"';' not allowed before 'ELSE'"

Examples Compiler error messages

Description

You have placed a ';' directly before an **else** in an **if..else** statement. The reason for this is that the ';' is treated as a statement separator, not a statement terminator -- **if..else** is one statement, a ';' cannot appear in the middle (unless you use compound statements).

Examples

{ Pascal does not allow a ';' to be placed directly before an ELSE statement. In the code above, an error will be flagged because of this fact. }

```
program Produce;

var
  B: Integer;

begin
  if B = 10 then
    B := 0;
  else
    B := 10;
end.
```

{ There are two easy solutions to this problem. The first is to remove the offending ';'. The second is to create compound statements for each part of the IF..ELSE. If $HINTS are turned on, you will receive a hint about the value assigned to B is never used. statement. }

```
program Solve;

var
  B: Integer;

begin
  if B = 10 then
    B := 0
  else
    B := 10;

  if B = 10 then
  begin
    B := 0;
  end
  else
  begin
    B := 10;
  end;

end.
```

"Type '<name>' needs finalization - not allowed in variant record"

Examples Compiler error messages

Description

Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at run time, it is not possible to guarantee that these special data types are correctly finalized.

Examples

{ String is one of those types which requires special treatment by the compiler to correctly release the resources. As such, it is illegal to have a String in a variant section. }

```
program Produce;

type
  Data = record
    case Kind:Char of
    'A': (Str: string);
  end;

begin
end.
```

{ One solution to this error is to move all offending declarations out of the variant section. Another solution would be to use pointer types (^String, for example) and manage the memory yourself. }

```
program Solve;

type
  Data = record
    Str: string;
  end;

begin
end.
```

"Type '<name>' needs finalization - not allowed in file type"

Examples Compiler error messages

Description

Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own.
Because the compiler cannot determine what the length of a long string will be at run time, it is not possible to guarantee that these special data types are correctly finalized. Records in typed files must all be of the same size, so types containing long strings are not allowed.

Examples

{ String is one of those data types which need finalization, and as such they cannot be stored in a File type. }

```
program Produce;

type
  Data = record
    Name: string;
  end;

var
  InFile: file of Data;

begin
end.
```

{ One simple solution, for the case of String, is to redeclare the type as an array of characters. For other cases which require finalization, it becomes increasingly difficult to maintain a binary file structure with standard Pascal features, such as 'file of'. In these situations, it is probably easier to write specialized file I/O routines. }

```
program Solve;

type
  Data = record
    Name: array[1..25] of Char;
  end;

var
  InFile: file of Data;

begin
end.
```

"Expression too complicated"

Compiler error messages

Description

The compiler has encounter an expression in your source code that is too complicated for it to handle. Reduce the complexity of your expression by introducing some temporary variables.

"Element 0 inaccessible - use 'Length' or 'SetLength'"

Examples Compiler error messages

Description

The Delphi **string** type does not store the length of the string in element 0. The old method of changing, or getting, the length of a string by accessing element 0 does not work with long strings.

Examples

{ Here the program is attempting to get the length of the string by directly accessing the first element. This is not legal. }

```
program Produce;

var
  Str: string;
  Len: Integer;

begin
  Str := 'Kojo no tsuki';
  Len := Str[0];
end.
```

{ You can use the SetLength and Length standard procedures to provide the same functionality as directly accessing the first element of the string. If hints are turned on, you will receive a warning about the value of 'len' not being used. }

```
program Solve;

var
  Str: string;
  Len: Integer;

begin
  Str := 'Kojo no tsuki';
  Len := Length(Str);
end.
```

"System unit out of date or corrupted: missing '<name>'"

Compiler error messages

Description

The compiler is looking for a special function which resides in System.dcu but could not find it. Your System unit is either corrupted or obsolete.
Make sure there are no conflicts in your library search path which can point to another System.dcu. Try reinstalling System.dcu. If neither of these solutions work, then contact Borland International.

"Record, object or class type required"

Examples Compiler error messages

Description

The compiler was expecing to find the type name which specified a record, object or class but did not find one. There are two causes for this error. The first is the application of '.' to an object that is not a record. The second cause is the use of a variable which is of the wrong type in a **with** statement.

Examples

{ There are two causes for the same error in this program. The first is the application of '.' to a object that is not a record. The second case is the use of a variable which is of the wrong type in a WITH statement. }

```
program Produce;

type
  RecordDesc = class
    Ch: Char;
  end;

var
  pCh: PChar;
  r: RecordDesc;

procedure A;
begin
  pCh.Ch := 'A';    (* case 1 *)
  with pCh do
  begin (* case 2 *)
  end;
end;
end.
```

{ The easy solution to this error is to always make sure that the '.' and WITH are both applied only to records, objects or class variables. }

```
program Solve;

type
  RecordDesc = class
    Ch: Char;
  end;

var
  R: RecordDesc;

procedure A;
begin
  R.Ch := 'A';    (* case 1 *)
  with r do
  begin (* case 2 *)
  end;
end;
end.
```

"Type not allowed in OLE Automation call"

Examples Compiler error messages

Description

If a data type cannot be converted by the compiler into a variant, then it is not allowed in an OLE automation call.

Examples

{ A class cannot be converted into a Variant type, so it is not allowed in an OLE call. }

```
program Produce;

type
  Base = class
    X: Integer;
  end;

var
  B: Base;
  V: Variant;

begin
  V.Dispatch(B);
end.
```

{ The only solution to this problem is to manually convert these data types to Variants or to only use data types that can automatically be converted into a Variant. }

```
program Solve;

type
  Base = class
    X: Integer;
  end;

var
  B: Base;
  V: Variant;

begin
  V.Dispatch(B.X);
end.
```

"<RLink32 error message>"

Compiler error messages

Description

RLINK32 has encountered an error, which it is duly reporting to you.
Please refer to the RLINK32 reference for a more thorough description of this error.

"<Filename>: <RLink32 error message>"

Compiler error messages

Description

RLINK32 has encountered an error, which it is duly reporting to you.
Please refer to the RLINK32 reference for a more thorough description of this error.

"Too many conditional symbols"

Compiler error messages

Description

You have exceeded the memory allocated to conditional symbols defined on the command line (including configuration files). There are 256 bytes allocated for all the conditional symbols. Each conditional symbol requires 1 extra byte when stored in conditional symbol area.
The only solution is to reduce the number of conditional compilation symbols contained on the command line (or in confiugration files).

"Method '<name>' hides virtual method of base type '<name>'"

See also Examples Compiler error messages

Description

You have declared a method which has the same name as a virtual method in the base class. Your new method is not a virtual method; it will hide access to the base's method of the same name.

Examples

{ Both methods declared in the definition of Derived will hide the virtual functions of the same name declared in the base class. }

```
program Produce;

type
  Base = class
    procedure VirtuMethod; virtual;
    procedure VirtuMethod2; virtual;
  end;

  Derived = class(Base)
    procedure VirtuMethod;
    procedure VirtuMethod2;
  end;

procedure Base.VirtuMethod;
begin
end;

procedure Base.VirtuMethod2;
begin
end;

procedure Derived.VirtuMethod;
begin
end;

procedure Derived.VirtuMethod2;
begin
end;

begin
end.
```

{ There are two alternatives to take when solving this error. First, you could specify override to make the derived class' procedure also virtual, and thus allowing inherited calls to still reference the original procedure. You could also change the name of the procedure as it is declared in the derived class. Both methods are exhibited in this example. }

```
program Solve;

type
  Base = class
    procedure VirtuMethod; virtual;
    procedure VirtuMethod2; virtual;
  end;

  Derived = class(Base)
    procedure VirtuMethod; override;
    procedure Virtu2Method;
  end;

procedure Base.VirtuMethod;
begin
end;
```

```
procedure Base.VirtuMethod2;
begin
end;

procedure Derived.VirtuMethod;
begin
end;

procedure Derived.Virtu2Method;
begin
end;

begin
end.
```

See also
Overriding Methods

"Variable '<name>' is declared but never used in '<name>'"

Examples Compiler error messages

Description

You have declared a variable in a procedure, but you never actually use it.

Examples

```
program Produce;
{$HINTS ON}

procedure Local;
var
  I: Integer;
begin
end;

begin
end.
```

{ One simple solution is to remove any unused variable from your procedures. However, unused variables can also indicate an error in the implementation of your algorithm. }

```
program Solve;

{$HINTS ON}

procedure Local;
begin
end;

begin
end.
```

"Compile terminated by user"

Compiler error messages

Description

You pressed the Cancel button in the Compiling dialog box during a compile.

"Unnamed arguments must precede named arguments in OLE Automation call"

Examples Compiler error messages

Description

You have attempted to follow named OLE Automation parameterswith unnamed parameters.

Examples

{ The named parameter FileName must follow the unnamed parameter in this OLE dispatch. }

```
program Produce;

var
  Ole: Variant;

begin
  Ole.Dispatch(FileName:='FrogEggs', 'Tapioca');
end.
```

{ This solution, reversing the parameters, is the most straightforward but it may not be appropriate for your situation. Another alternative would be to provide the unnamed parameter with a name. }

```
program Solve;

var
  Ole: Variant;

begin
  Ole.dispatch('Tapioca', FileName:='FrogEggs');
end.
```

"Abstract methods must be virtual or dynamic"

Examples Compiler error messages

Description

When declaring an abstract method in a base class, it must either be of regular virtual or dynamic virtual type.

Examples

{ The declaration that follows is in error because abstract methods must either be virtual or dynamic. }

```
program Produce;

type
  Base = class
    procedure DaliVision; abstract;
    procedure TellyVision; abstract;
  end;

begin
end.
```

{ It is possible to remove this error by either specifying virtual or dynamic, whichever is most appropriate for your application. }

```
program Solve;

type
  Base = class
    procedure DaliVision; virtual; abstract;
    procedure TellyVision; dynamic; abstract;
  end;

begin
end.
```

"Case label outside of range of case expression"

Examples Compiler error messages

Description

You have provided a label inside a **case** statement which cannot be produced by the case statement control variable.

Examples

{ It is not possible for a TatesCompass to hold all the values of the CompassPoints, and so several of the case labels will elict errors. }

```
program Produce;
{$WARNINGS ON}

type
  CompassPoints = (n, e, s, w, ne, se, sw, nw);
  FourPoints = n..w;

var
  TatesCompass: FourPoints;

begin
  TatesCompass := e;
  case TatesCompass of
    n:    Writeln('North');
    e:    Writeln('East');
    s:    Writeln('West');
    w:    Writeln('South');
    ne:   Writeln('Northeast');
    se:   Writeln('Southeast');
    sw:   Writeln('Southwest');
    nw:   Writeln('Northwest');
  end;
end.
```

{ After examining your code to determine what the intention was, there are two alternatives. The first is to change the type of the case statement's control variable so that it can produce all the case labels. The second alternative would be to remove any case labels that cannot be produced by the control variable. The first alternative is shown in this example. }

```
program Solve;
{$WARNINGS ON}

type
  CompassPoints = (n, e, s, w, ne, se, sw, nw);
  FourPoints = n..w;

var
  TatesCompass: CompassPoints;

begin
  TatesCompass := e;
  case TatesCompass OF
    n:    Writeln('North');
    e:    Writeln('East');
    s:    Writeln('West');
    w:    Writeln('South');
    ne:   Writeln('Northeast');
    se:   Writeln('Southeast');
    sw:   Writeln('Southwest');
    nw:   Writeln('Northwest');
  end;
end.
```

"Object type required"

Examples Compiler error messages

Description

This error is given whenever an object type is expected by the compiler. For instance, the ancestor type of an object must also be an object type.

Examples

{ TObject in the System unit is a class type, so we cannot derive an object type from it. }

```
type
  MyObject = object(TObject)
  end;
begin
end.
```

{ Make sure the type identifier really stands for an object type - maybe it is misspelled, or maybe is hidden by an identifier from another unit. }

```
program Solve;
type
  MyObject = class  { Actually, this means: class(TObject) }
  end;
begin
end.
```

"Field or method identifier expected"

Examples Compiler error messages

Description

You have specified an identifier for a **read** or **write** clause to a property which is not a field or method.

Examples

{ The two properties in this code both cause errors. The first causes an error because it is not possible to specify the property itself as the read & write methods. The second causes an error because 'r' is not a member of the Base class. }

```
program Produce;

var
  R: string;

type
  Base = class
    T: string;
    property Title: string read Title write Title;
    property Caption: string read R write R;
  end;

begin
end.
```

{ To solve this error, make sure that all read and write clauses for properties specify a valid field or method identifier that is a member of the class which owns the property. }

```
program Solve;

type
  Base = class
    T: string;
    property Title: string read T write T;
  end;

begin
end.
```

"Constructing instance of '<name>' containing abstract methods"

Examples Compiler error messages

Description

The code you are compiling is constructing instances of classes which contain abstract methods.

Examples

{ An abstract procedure does not exist, so it becomes dangerous to create instances of a class which contains abstract procedures. In this case, the creation of 'b' is the cause of the warning. Any invocation of 'Abstraction' through the instance of 'b' created here would cause a runtime error. A hint will be issued that the value assigned to 'b' is never used. }

```
program Produce;
{$WARNINGS ON}
{$HINTS ON}

type
  Base = class
    procedure Abstraction; virtual; abstract;
  end;

var
  B: Base;

begin
  B := Base.Create;
end.
```

{ One solution to this problem is to remove the abstract directive from the procedure declaration, as is shown here. Another method of approaching the problem would be to derive a class from Base and then provide a concrete version of Abstraction. A hint will be issued that the value assigned to B is never used. }

```
program Solve;
{$WARNINGS ON}
{$HINTS ON}

type
  Base = class
    procedure Abstraction; virtual;
  end;

var
  B: Base;

procedure Base.Abstraction;
begin
end;

begin
  B := Base.Create;
end.
```

"Field definition not allowed after methods or properties"

Examples Compiler error messages

Description

You have attempted to add more fields to a class after the first method or property declaration has been encountered. You must place all field definitions before methods and properties.

Examples

{ The declaration of 'a' after 'FirstMethod' will cause an error. }

```
program Produce;

type
  Base = class
   procedure FirstMethod;
    A: Integer;
  end;


procedure Base.FirstMethod;
begin
end;

begin
end.
```

{ To solve this error, it is normally sufficient to simply move all field definitions before the first field or property declaration. }

```
program Solve;

type
  Base = class
    A: Integer;
    procedure FirstMethod;
  end;


procedure Base.FirstMethod;
begin
end;

begin
end.
```

"Cannot override a static method"

Examples Compiler error messages

Description

You have tried, in a derived class, to override a base method which was not declared as one of the virtual types.

Examples

{ The example above elicits an error because Base.StaticMethod is not declared to be a virtual method, and as such it is not possible to override its declaration. }

```
program Produce;

type
  Base = class
    procedure StaticMethod;
  end;

  Derived = class(Base)
    procedure StaticMethod; override;
  end;

procedure Base.StaticMethod;
begin
end;

procedure Derived.StaticMethod;
begin
end;

begin
end.
```

{ The only way to remove this error from your program, when you don't have the source for the base classes, is to remove the 'override' specification from the declaration of the derived method. If you have source to the base classes, you could, with careful consideration, change the base's method to be declared as one of the virtual types -- but be aware that this change can have a drastic affect on your programs. }

```
program Solve;

type
  Base = class
    procedure StaticMethod;
  end;

  Derived = class(Base)
    procedure StaticMethod;
  end;

procedure Base.StaticMethod;
begin
end;

procedure Derived.StaticMethod;
begin
end;

begin
end.
```

"Variable '<name>' inaccessible here due to optimization"

Examples Compiler error messages

Description

The evaluator or watch statement is attempting to retrieve the value of <name>, but the compiler was able to determine that the variables actual lifetime ended prior to this inspection point. This error will often occur if the compiler determines a local variable is assigned a value that is not used beyond a specific point in the program's control flow.

Examples

1. Create a new application.
2. Place a button on the form.
3. Double-click the button to be taken to the 'click' method.
4. Add a global variable, C, of type Integer to the implementation section.

   The click method should read as:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  A, B: Integer;
begin
  A := 10;
  B := 20;
  C := B;
  A := C;
end;
```

5. Set a breakpoint on the assignment to C.
6. Compile and run the application.
7. Press the button.
8. After the breakpoint is reached, open the evaluator (Run|Evaluate/Watch).
9. Evaluate A.

The compiler realizes that the first assignment to A is dead, since the value is never used. As such, it defers even using A until the second assignment occurs -- up until the point where C is assigned to A, the variable A is considered to be dead and cannot be used by the evaluator.
The only solution is to only attempt to view variables which are known to have live values.

"Necessary library helper function was eliminated by linker"

Examples Compiler error messages

Description

The integrated debugger is attempting to use some of the compiler helper functions to perform the requested evaluate. The linker, on the other hand, determined that the helper function was not actually used by the program and it did not link it into the program.

Examples

1 Create a new application.
2 Place a button on the form.
3 Double click the button to be taken to the 'click' method.
4 Add a global variable, V, of type **string** to the interface section.
5 Add a global variable, P, of type PChar to the interface section.

The click method should read as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  V := 'Initialized';
  P := nil;
  V := 'Abid';
end;
```

6 Set a breakpoint on the second assignment to V.
7 Compile and run the application.
8 Press the button.
9 After the breakpoint is reached, open the evaluator (Run|Evaluate/Watch).
10 Evaluate V.
11 Move the cursor to the 'New Value' box.
12 Type in P.
13 Choose Modify.

The compiler uses a special function to copy a PChar to a string. In order to reduce the size of the produced executable, if that special function is not used by the program, it is not linked in. In this case, there is no assignment of a PChar to a string, so it is eliminated by the linker.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  V := 'Initialized';
  P := nil;
  V := 'Abid';
  V := p;
end;
```

Adding the extra assignment of a PChar to a string will ensure that the linker includes the desired procedure in the program. Encountering this error during a debugging session is an indicator that you are using some language/environment feature that was not needed in the original program.

"Missing or invalid conditional symbol in '$<symbol>' directive"

Examples Compiler error messages

Description

The **$IFDEF**, **$IFNDEF**, **$DEFINE** and **$UNDEF** directives require that a symbol follow them.

Examples

{ The $IFDEF conditional directive is incorrectly specified here and will result in an error. }

```
program Produce;

{$IFDEF}
{$ENDIF}

begin
end.
```

{ The solution to the problem is to ensure that a symbol to test follows the appropriate directives. }

```
program Solve;

{$IFDEF WIN32}
{$ENDIF}

begin
end.
```

"Incompatible format specification"

Compiler error messages

Description

You have specified a format specifier for a watch or evaluate statement that is incompatible with the type of the object which you are inspecting. For example, attempting to display a Boolean variable as a string will result in this error.

"Format specifier must be C, S, D, H, X, Fn, P, R or nM"

Compiler error messages

Description

You have attempted to specify a format specifier for the watch/evaluate expression which is invalid. You must specify a valid format specification before the evaluation is able to proceeed.

"Bad specification of M format"

Compiler error messages

Description

You have specified an M format specifier which is not allowed. Only C, D, H, X, S, or M are allowed as suffixes to the M specifier.

"Object or class type required"

Examples Compiler error messages

Description

This error message is given when the syntax 'Typename.Methodname' is used, but the typename does not refer to an object or class type.

Examples

{ Type Integer does not have a Create method of course - TInteger does. }

```
program Produce;
type
  TInteger = class
    Value: Integer;
  end;
var
  V: TInteger;
begin
  V := Integer.Create;
end.
```

{ Make sure the identifier really refers to an object or class type - maybe it is misspelled or it is hidden by an identifier from another unit. }

```
program Solve;
type
  TInteger = class
    Value: Integer;
  end;
var
  V: TInteger;
begin
  V := TInteger.Create;
end.
```

"Fn requires 2 <= n <= 18"

Examples Compiler error messages

Description

You have specified a value which is out of range for the floating point format specifier in the integrated debugger. The range for the Fn format specifier is 2..18.

Examples

1. Create an application with a real variable.
2. Evaluate that variable with ',F21' on the end of the evaluate command (such as: realVar,F21).
3. Reevaluate f, but specify a number in the range 2..18 for the Fn specifier.

When evaluating real numbers with the Fn specification, make sure 2 <= n <= 18.

"'<name>' not previously declared as a PROPERTY"

Examples Compiler error messages

Description

You have attempted to hoist a property to a different visibility level by redeclaration, but <name> in the base class was not declared as a property.

Examples

{ The intent of the redeclaration of 'Derived.Title' is to change the field which is used to read and write the property 'Title' as well as hoist it to 'public' visibility. Unfortunately, the programmer really meant to use 'TitleProp', not 'Title'. }

```
program Produce;
{$WARNINGS ON}

type
  Base = class
  protected
    Caption: string;
    Title: string;
    property TitleProp: string read Title write Title;
  end;

  Derived = class(Base)
  public
    property Title read Caption write Caption;
  end;

begin
end.
```

{ There are a couple ways of approaching this error. The first, and probably the most commonly taken, is to specify the real property which is to be redeclared. The second, which can be seen in the rececleration of 'Title' addresses the problem by explicitly creating a new property, with the same name as a field in the base class. This new property will hide the base field, which will no longer be accessible without a typecast. (Note: If you have warnings turned on, the redeclaration of 'Title' will issue a warning notifying you that the redeclaration will hide the base class' member.) }

```
program Solve;
{$WARNINGS ON}

type
  Base = class
  protected
    Caption: string;
    Title: string;
    property TitleProp: string read Title write Title;
  end;

  Derived = class(Base)
  public
    property TitleProp read Caption write Caption;
    property Title: string read Caption write Caption;
  end;

begin
end.
```

"Field definition not allowed in OLE automation section"

Examples Compiler error messages

Description

You have tried to place a field definition in an OLE automation section of a class declaration. Only properties and methods may be declared in an **automated** section.

Examples

{ The declaration of I in this class will cause the compile error. }

```
program Produce;

type
  Base = class
  automated
    I: Integer;
  end;

begin
end.
```

{ Moving the declaration of I out of the automated section will vanquish the error. }

```
program Solve;

type
  Base = class
    I: Integer;
  automated
  end;

begin
end.
```

"Illegal type in OLE automation section: '<typename>'"

Examples Compiler error messages

Description

<typename> is not an allowed type in an **automated** section. Only a small subset of all the valid Pascal types are allowed in automation sections.

Examples

{ Since the character type is not one allowed in the 'automated' section, the declaration of 'Ch' will produce an error when compiled. }

```
program Produce;

type
  Base = class
    function GetC: Char;
    procedure SetC(C: Char);
  automated
    property Ch: Char read GetC write SetC dispid 151;
  end;

procedure Base.SetC(C: Char);
begin
end;

function Base.GetC: Char;
begin
  GetC := '!';
end;

begin
end.
```

{ There are two solutions to this problem. The first is to move the offending declaration out of the 'automated' section. The second is to change the offending type to one that is allowed in 'automated' sections. }

```
program Solve;

type
  Base = class
    function GetC: string;
    procedure SetC(C: string);
  automated
    property Ch: string read GetC write SetC dispid 151;
  end;

procedure Base.SetC(C: string);
begin
end;

function Base.GetC: string;
begin
  GetC := '!';
end;

begin
end.
```

"String constant truncated to fit STRING[<number>]"

See also Examples Compiler error messages

Description

A string constant is being assigned to a variable which is not large enough to contain the entire string. The compiler is alerting you to the fact that it is truncating the literal to fit into the variable.

Examples

{ The two string constants are assigned to variables which are too short to contain the entire string. The compiler will truncate the strings and perform the assignment. }

```
program Produce;
{$WARNINGS ON}

const
  Title = 'Super Galactic Invaders with Turbo Gungla Sticks';
  Subtitle = 'Copyright (c) 1968 by Frank Borland';

type
  TitleString = string[25];
  SubtitleString = string[18];

var
  ProgramTitle: TitleString;
  ProgramSubtitle: SubtitleString;

begin
  ProgramTitle := Title;
  ProgramSubtitle := Subtitle;
end.
```

{ There are two solutions to this problem, both of which are demonstrated in this example. The first solution is to increase the size of the variable to hold the string. The second is to reduce the size of the string to fit in the declared size of the variable. }

```
program Solve;
{$WARNINGS ON}

const
  Title = 'Super Galactic Invaders with Turbo Gungla Sticks';
  Subtitle = 'Copyright (c) 1968';

type
  TitleString = string[55];
  SubtitleString = string[18];

var
  ProgramTitle: TitleString;
  ProgramSubtitle: SubtitleString;

begin
  ProgramTitle := Title;
  ProgramSubtitle := Subtitle;
end.
```

See also
Short string types

"Constructors and destructors not allowed in OLE automation section"

Examples Compiler error messages

Description

You have incorrectly tried to put a constructor or destructor into the automated section of a class declaration.

Examples

{ It is not possible to declare a class constructor or destruction in an OLE automation section. The constructor and destructor declarations in the above code will both elicit this error. }

```
program Produce;

type
  Base = class
  automated
    constructor HardHatBob;
    destructor  DemolitionBob;
  end;

constructor Base.HardHatBob;
begin
end;

destructor Base.DemolitionBob;
begin
end;

begin
end.
```

{ The only solution to this error is to move your declarations out of the automated section, as has been done in this example. }

```
program Solve;

type
  Base = class
    constructor HardHatBob;
    destructor  DemolitionBob;
  end;

constructor Base.HardHatBob;
begin
end;

destructor Base.DemolitionBob;
begin
end;

begin
end.
```

"Dynamic methods and message handlers not allowed in OLE automation section"

Examples Compiler error messages

Description

You have incorrectly put a dynamic or message method into an automated section of a class declaration.

Examples

{ It is not possible to have a dynamic or message method declaration in an OLE automation section of a class. As such, the two method declarations in the above program both produce errors. }

```
program Produce;

type
  Base = class
  automated
    procedure DynaMethod; dynamic;
    procedure MessageMethod(var Msg: Integer); message 151;
  end;

procedure Base.DynaMethod;
begin
end;

procedure Base.MessageMethod;
begin
end;

begin
end.
```

{ There are several ways to remove this error from your program. First, you could move any declaration which produces this error out of the automated section, as has been done in this example. Alternatively, you could remove the dynamic or message attributes of the method; of course, removing these attributes will not provide you with the desired behavior, but it will remove the error. }

```
program Solve;

type
  Base = class
    procedure DynaMethod; dynamic;
    procedure MessageMethod(var Msg: Integer); message 151;
  end;

procedure Base.DynaMethod;
begin
end;

procedure Base.MessageMethod;
begin
end;

begin
end.
```

"Only register calling convention allowed in OLE automation section"

Examples Compiler error messages

Description

You have specified an illegal calling convention on a method appearing in an **automated** section of a class declaration.

Examples

{ The language specification disallows all calling conventions except 'register' in an OLE automation section. The offending statement is 'cdecl' in the following code. }

```
program Produce;

type
  Base = class
  automated
    procedure Method; cdecl;
  end;

procedure Base.Method; cdecl;
begin
end;

begin
end.
```

{ There are three solutions to this error. The first is to specify no calling convention on methods declared in an auto section. The second is to specify only the register calling convention. The third is to move the offending declaration out of the automation section. }

```
program Solve;

type
  Base = class
  automated
    procedure Method; register;
    procedure Method2;
  end;

procedure Base.Method; register;
begin
end;

procedure Base.Method2;
begin
end;

begin
end.
```

"Dispid '<number>' already used by '<name>'"

See also Examples Compiler error messages

Description

An attempt to use a **dispid** which is already assigned to another member of this class.

Examples

{ Each automated property's dispid must be unique, thus SecondValue is in error. }

```
program Produce;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
  automated
    property Value: Integer read GetV write SetV dispid 151;
    property SecondValue: Integer read GetV write SetV dispid 151;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

{ Giving a unique dispid to SecondValue will remove the error. }

```
program Solve;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
  automated
    property Value: Integer read getV write setV dispid 151;
    property SecondValue: Integer read GetV write SetV dispid 152;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

See also

OLE Automation

Properties

"Redeclaration of property not allowed in OLE automation section"

Examples Compiler error messages

Description

It is not allowed to redeclare a property into an automated section.

Examples

{ In the following example, Name is moved from a private visibility in Base to public visibility in Derived by redeclaration. The same idea is attempted on Value, but an error results. }

```
program Produce;

type
  Base = class
    V: Integer;
    S: string;
  protected
    property Name: string read S write S;
    property Value: Integer read V write V;
  end;

  Derived = class(Base)
  public
    property Name; (* Move Name to a public visibility by redeclaration *)
  automated
    property Value;
  end;

begin
end.
```

{ It is simply not possible to change the visibility of a property to an automated section, therefore the solution to this problem is to not redeclare properties of base classes in automated sections. }

```
program Solve;

type
  Base = class
    V: Integer;
    S: string;
  protected
    property Name: string read S write S;
    property Value: Integer read V write V;
  end;

  Derived = class(Base)
  public
    property Name; (* Move Name to a public visibility by redeclaration *)
    property Value;
  automated
  end;

begin
end.
```

"Undeclared identifier: '<name>'"

Examples Compiler error messages

Description

The compiler could not find the given identifier - most likely it has been misspelled either at the point of declaration or the point of use. It might be from another unit that has not mentioned a **uses** clause.

Examples

{ In the example the variable has been declared as "Counter", but used as "Count". The solution is to either change the declaration or the places where the variable is used. }

```
program Produce;
var
  Counter: Integer;
begin
  Count := 0;
  Inc(Count);
  Writeln(Count);
end.
```

{ In the example we have chosen to change the declaration - that was less work. }

```
program Solve;
var
  Count: Integer;
begin
  Count := 0;
  Inc(Count);
  Writeln(Count);
end.
```

"Class type required"

Examples Compiler error messages

Description

In certain situations the compiler requires a class type:

- As the ancestor of a class type
- In the **on** clause of a **try**..**except** statement
- As the first argument of a **raise** statement
- As the final type of a forward-declared class type

Examples

```
program Produce;
begin
  raise 'This would work in C++, but does not in Delphi';
end.

program Solve;
uses SysUtils;
begin
  raise Exception.Create('There is a simple workaround, however');
end.
```

"'<clause>' clause not allowed in OLE automation section"

Examples Compiler error messages

Description

The directives **index**, **stored**, **default**, and **nodefault** are not allowed in OLE automation sections.

Examples

{ Including a NODEFAULT clause on an automated property is not allowed. }

```
program Produce;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
  automated
    property Value: Integer read GetV write SetV nodefault;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

{ Simply removing the offending clause will cause the error to go away. Alternatively, moving the property out of the automated section will also make the error go away. }

```
program Solve;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
  automated
    property Value: Integer read GetV write SetV;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

"Dispid clause only allowed in OLE automation section"

Description

A **dispid** has been given to a property which is not in an **automated** section.

Examples

{ This program attempts to set the dispid for an OLE automation object, but the property has not been declared in an automated section. }

```
program Produce;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
    property Value: Integer read GetV write SetV dispid 151;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

{ To solve the error, you can either remove the dispid clause from the property declaration, or move the property declaration into an automated section. }

```
program Solve;

type
  Base = class
    V: Integer;
    procedure SetV(X: Integer);
    function GetV: Integer;
  automated
    property Value: Integer read GetV write SetV dispid 151;
  end;

procedure Base.SetV(X: Integer);
begin
  V := X;
end;

function Base.GetV: Integer;
begin
  GetV := V;
end;

begin
end.
```

"Type '<name>' must be a class to have OLE automation"

Examples Compiler error messages

Description

Old-style objects cannot have an automated section.

Examples

{ It is not possible to have an automated section in an old-style object, thus an error will result from this example. }

```
program Produce;

type
  OldObject = object
  automated
  end;

begin
end.
```

{ Changing the type from 'object' to 'class', or removing the automated section will remove the error. }

```
program Solve;

type
  NewClass = class
  automated
  end;

begin
end.
```

"Type '<name>' must be a class to have a PUBLISHED section"

Examples Compiler error messages

Description

Old-style objects cannot have a **published** section.

Examples

{ It is not possible to have a published section in an old-style object, thus an error will result from this example. }

```
{$TYPEINFO ON}
program Produce;

type
  OldObject = object
  published
  end;

begin
end.
```

{ Changing the type from 'object' to 'class', or removing the published section will remove the error. }

```
{$TYPEINFO ON}
program Solve;

type
  NewClass = class
  published
  end;

begin
end.
```

"Redeclaration of '<name>' hides a member in the base class"

Examples Compiler error messages

Description

A property has been created in a class with the same name of a variable contained in one of the base classes. One possible, and not altogether apparent, reason for getting this error is that a new version of the base class hierarchy has been installed and it contains new member variables which have names identical to your properties' names.

Examples

{ Derived.v overrides, and thus hides, Base.v; it will not be possible to access Base.v in any variable of type Derived without a typecast. }

```
{$WARNINGS ON}
program Produce;

type
  Base = class
    V: Integer;
  end;

  Derived = class(Base)
    Ch: Char;
    property V: Char read Ch write Ch;
  end;

begin
end.
```

{ By simply changing the name of the property in the derived class, the error is alleviated. }

```
{$WARNINGS ON}
program Solve;
type
  Base = class
    V: Integer;
  end;

  Derived = class(Base)
    Ch: Char;
    property ChV: Char read Ch write Ch;
  end;

begin
end.
```

"Overriding automated virtual method '<name>' cannot specify a dispid"

Examples Compiler error messages

Description

The **dispid** declared for the original virtual **automated** procedure declaration must be used by all overriding procedures in derived classes.

Examples

{ The overriding declaration of Base.Automatic, in Derived (Derived.Automatic) erroneously attempts to define another dispid for the procedure. }

```
program Produce;

type
  Base = class
  automated
    procedure Automatic; virtual; dispid 151;
  end;


  Derived = class(Base)
  automated
    procedure Automatic; override; dispid 152;
  end;

procedure Base.Automatic;
begin
end;

procedure Derived.Automatic;
begin
end;

begin
end.
```

{ By removing the offending dispid clause, the program will now compile. }

```
program Solve;

type
  Base = class
  automated
    procedure Automatic; virtual; dispid 151;
  end;

  Derived = class(Base)
  automated
    procedure Automatic; override;
  end;

procedure Base.Automatic;
begin
end;

procedure Derived.Automatic;
begin
end;

begin
end.
```

"Published Real property '<name>' must be Single, Double or Extended"

Examples Compiler error messages

Description

You have attempted to publish a property of type Real, which is not allowed. Published floating point properties must be Single, Double or Extended.

Examples

{ The published Real property in the program above must be either removed, moved to an unpublished section or changed into an acceptable type. }

```
program Produce;
type
  Base = class
    R: Real;
  published
    property RVal: Real read R write R;
  end;
end.
```

{ This solution changed the property into a real type that will actually produce run-time type information. }

```
program Produce;
type
  Base = class
    R: Single;
  published
    property RVal: Single read R write R;
  end;
end.
```

"Size of published set '<name>' is >32 bits"

Examples Compiler error messages

Description

The compiler does not allow sets greater than 32 bits to be contained in a published section. The size, in bytes, of a set can be calculated by High(setname) div 8 - Low(setname) div 8 + 1.

Examples

```
{$TYPEINFO ON}
program Produce;
type
  CharSet = set of Char;
  NamePlate = class
    Characters: CharSet;
  published
    property TooBig: CharSet read Characters write Characters ;
  end;

begin
end.

{$TYPEINFO ON}
program Solve;
type
  CharSet = set of 'A'..'Z';
  NamePlate = class
    Characters: CharSet;
  published
    property TooBig: CharSet read Characters write Characters ;
  end;

begin
end.
```

"Published property '<name>' cannot be of type <type>"

Examples Compiler error messages

Description

Published properties must be an ordinal type, Single, Double, Extended, Comp, a string type, a set type which fits in 32 bits, or a method pointer type. When any other property type is encountered in a published section, the compiler will remove the published attribute

Examples

{ An error is induced because an array is not one of the data types which can be published. }

```
{$TYPEINFO ON}
program Produce;

type
  TitleArr = array[0..24] of Char;
  NamePlate = class
    TitleStr: TitleArr;
  published
    property Title: TitleArr read TitleStr write TitleStr;
  end;

begin
end.
```

{ Moving the property declaration out of the published section will avoid this error. Another alternative, as in this example, is to change the type of the property to be something that can actually be published. }

```
{$TYPEINFO ON}
program Solve;

type
  TitleArr = Integer;
  NamePlate = class
    TitleStr: TitleArr;
  published
    property Title: TitleArr read TitleStr write TitleStr;
  end;

begin
end.
```

"Thread local variables cannot be local to a function"

Examples Compiler error messages

Description

Thread-local variables must be declared at a global scope.

Examples

{ A thread variable cannot be declared local to a procedure. }

```
program Produce;

procedure NoTLS;
threadvar
  X: Integer;
begin
end;

begin
end.
```

{ There are two simple alternatives for avoiding this error. First, the threadvar section can be moved to a local scope. Secondly, the theadvar in the procedure could be changed into a normal var section. Note that if compiler hints are turned on, a hint about localX being declared but not used will be emitted. }

```
program Solve;

threadvar
  X: Integer;

procedure YesTLS;
var
  LocalX: Integer;
begin
end;

begin
end.
```

"Function needs result type"

Examples Compiler error messages

Description

You have declared a function, but have not specified a return type.

Examples

{ Here Sum is meant to be function, we have not told the compiler about it. }

```
program Produce;

function Sum(A: array of Integer);
var
  I: Integer;
begin
  Result := 0;
  for I := 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

{ Just make sure you specify the result type. }

```
program Solve;

function Sum(A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

"Thread local variables cannot be ABSOLUTE"

Examples Compiler error messages

Description

A thread local variable cannot refer to another variable, nor can it reference an absolute memory address.

Examples

{ The absolute directive is not allowed in a threadvar declaration section. }

```
program Produce;

threadvar
  SecretNum: Integer absolute $151;

begin
end.
```

{ There are two easy ways to solve a problem of this nature. The first is to simply remove the absolute directive from the threadvar section. The second would be to move the absolute variable to a normal var declaration section. }

```
program Solve;

threadvar
  SecretNum: Integer;

var
  sNum: Integer absolute $151;

begin
end.
```

"EXPORTS allowed only at global scope"

Examples Compiler error messages

Description

An **exports** clause has been encountered in the program source at a non-global scope.

Examples

{ It is not allowed to have an EXPORTS clause anywhere but a global scope. }

```
program Produce;

procedure ExportedProcedure;
exports ExportedProcedure;
begin
end;

begin
end.
```

{ The solution is to ensure that your EXPORTS clause is at a global scope and textually follows all procedures named in the clause. As a general rule, EXPORTS clauses are best placed right before the source file's initialization code. }

```
program Solve;

procedure ExportedProcedure;
begin
end;

exports ExportedProcedure;
begin
end.
```

"Constants cannot be used as open array arguments"

Examples Compiler error messages

Description

Open array parameters must be supplied with an actual array variable, a constructed array or a single variable of the parameter's element type.

Examples

{ The error is caused in this example because a string literal is being supplied when an array is expected. It is not possible to implicitly construct an array from a constant. }

```
program Produce;

procedure TakesArray(S: array of string);
begin
end;

begin
  TakesArray('Hello Error');
end.
```

{ The solution avoids the error because the array is explicitly constructed. }

```
program Solve;

procedure TakesArray(S: array of string);
begin
end;


begin
  TakesArray(['Hello Error']);
end.
```

"Slice standard function only allowed as open array argument"

Examples Compiler error messages

Description

An attempt has been made to pass an array slice to a fixed size array. Array slices can only be sent to open array parameters.

Examples

{ In the following example, the error is produced because TakesArray expects a fixed size array. }

```
program Produce;

type
  IntegerArray = array[1..10] of Integer;

var
  SliceMe: array[1..200] of Integer;

procedure TakesArray(X: IntegerArray);
begin
end;

begin
  TakesArray(Slice(SliceMe, 5));
end.
```

{ In the following example, the error is not produced because TakesArray takes an open array as the parameter. }

```
program Solve;

type
  IntegerArray = array[1..10] of Integer;

var
  SliceMe: array[1..200] of Integer;

procedure TakesArray(X: array of Integer);
begin
end;

begin
  TakesArray(Slice(SliceMe, 5));
end.
```

"Cannot initialize thread local variables"

Examples Compiler error messages

Description

The compiler does not allow initialization of thread-local variables.

Examples

{ The declaration and initialization of 'tls' above is not allowed. }

```
program Produce;

threadvar
  tls: Integer = 151;

begin
end.
```

{ You can declare thread local storage as normal, and then initialize it in the initialization section of your source file. }

```
program Solve;

threadvar
  tls: Integer;

begin
  tls := 151;
end.
```

"Cannot initialize local variables"

Examples Compiler error messages

Description

The compiler disallows the use of local initialized variables.

Examples

{ The declaration and initialization of i in procedure Show is illegal. }

```
program Produce;

var
  J: Integer;

procedure Show;
var
  I: Integer = 151;
begin
end;

begin
end.
```

{ You can use a programmatic style to set all variables to known values. }

```
program Solve;

var
  J: Integer;

procedure Show;
var
  I: Integer;
begin
  I := 151;
end;

begin
  J := 0;
end.
```

"Cannot initialize multiple variables"

Examples Compiler error messages

Description

Variable initialization can only occur when variables are declared individually.

Examples

{ The compiler will disallow the declaration and initialization of more than one variable at a time. }

```
program Produce;

var
  I, J: Integer = 151, 152;

begin
end.
```

{ Simple declare each variable by itself to allow initialization. }

```
program Solve;

var
  I: Integer = 151;
  J: Integer = 152;

begin
end.
```

"Constant object cannot be passed as var parameter"

Examples Compiler error messages

Description

As variable parameters are intended to be modified by the called procedure or function, you can not pass a constant object to a variable parameter. If your intention is just to pass a big datastructure efficiently, and the called function should not modify it, you can use a const parameter instead.

Examples

{ In the example, function has a variable parameter, but we are passing a constant to it. }

```
program Produce;
{$AppType Console}

function Max(var A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
       Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );   { <-- Error message here }
end.
```

{ The solution is to declare the parameter as a constant parameter (we do not intend to modify it, after all). Alternatively, you can also modify the call so it does not pass constants. }

```
program Solve;
{$AppType Console}

function Max(const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
      Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );
end.
```

"Invalid function result type"

Examples Compiler error messages

Description

File types are not allowed as function result types.

Examples

{ You cannot return a file from a function. }

```
program Produce;

function OpenFile(Name: string): file;
begin
end;

begin
end.
```

{ You can "return" the file as a variable parameter. Alternatively, you can allocate a file dynamically and return a pointer to it. }

```
program Solve;

procedure OpenFile(Name: string; var F: file);
begin
end;

begin
end.
```

"Procedure cannot have a result type"

Examples Compiler error messages

Description

You have declared a procedure, but given it a result type. Either you really meant to declare a function, or you should delete the result type.

Examples

{ Here DotProduct was really meant to be a function, we just happened to use the wrong keyword... }

```
program Produce;

procedure DotProduct(const A,B: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := 0 to High(A) do
    Result := Result + A[I]*B[I];
end;

const
  C: array[1..3] of Double = (1,2,3);

begin
  Writeln( DotProduct(C,C) );
end.
```

{ Just make sure you specify a result type when you declare a function, and no result type when you declare a procedure. }

```
program Solve;

function DotProduct(const A,B: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := 0 to High(A) do
    Result := Result + A[I]*B[I];
end;

const
  C: array[1..3] of Double = (1,2,3);

begin
  Writeln( DotProduct(C,C) );
end.
```

"Text after final 'END.' - ignored by compiler"

Examples Compiler error messages

Description

This warning is given when there is still source text after the final **end** and the period that constitute the logical end of the program. Possibly the nesting of begin..end is inconsistent (there is one 'end' too many somewhere). Check whether you intended the source text to be ignored by the compiler - maybe it is actually quite important.

Examples

```
program Produce;

begin
end.

Text here is ignored by Delphi 16-bit - Delphi 32-bit gives a warning.

//-------------------------------
program Solve;

begin
end.

{ You can of course always put text in comments - }
{  that will of course not cause a warning        }
```

"Constant expression expected"

Examples Compiler error messages

Description

The compiler expected a constant expression here, but the expression it found turned out not to be constant.

Examples

{ The call to Pos is not a constant expression to the compiler, even though its arguments are constants, and it could in principle be evaluated at compile time. }

```
program Produce;
const
  Message = 'Hello World!';
  WPosition = Pos('W', Message);
begin
end.
```

{ So in this case, we just have to calculate the right value for WPosition ourselves. }

```
program Solve;
const
  Message = 'Hello World!';
  WPosition = 7;
begin
end.
```

"Constant expression violates subrange bounds"

Examples Compiler error messages

Description

This error message occurs when the compiler can determine that a constant is outside the legal range. This can occur for instance if you assign a constant to a variable of a subrange type.

Examples

```
program Produce;
var
  Digit: 1..9;
begin
  Digit := 0;     { Get message: Constant expression violates subrange bounds
  }
end.

program Solve;
var
  Digit: 0..9;
begin
  Digit := 0;
end.
```

"Duplicate tag value"

Examples Compiler error messages

Description

This error message is given when a constant appears more than once in the declaration of a variant record.

Examples

```
program Produce;
type
  VariantRecord = record
    case Integer of
    0: (IntField: Integer);
    0: (RealField: Real);     { <-- Error message here }
  end;

begin
end.
```

```
program Solve;
type
  VariantRecord = record
    case Integer of
    0: (IntField: Integer);
    1: (RealField: Real);
  end;

begin
end.
```

"Sets may have at most 256 elements"

Examples Compiler error messages

Description

This error message appears when you try to declare a set type of more than 256 elements. More precisely, the ordinal values of the upper and lower bounds of the base type must be within the range 0..255.

Examples

{ In the example, BigSet really only has 256 elements, but is still illegal. }

```
program Produce;
type
  BigSet = set of 1..256;  { <-- error message given here }
begin
end.
```

{ We need to make sure the upper and lower bounds and in the range 0..255. }

```
program Solve;
type
  BigSet = set of 0..255;
begin
end.
```

"<Token1> expected but <token2> found"

Examples Compiler error messages

Description

This error message appears for syntax errors. There is probably a typo in the source, or something was left out. When the error occurs at the beginning of a line, the actual error is often on the previous line.

Examples

{ After the type Integer, the compiler expects to find a semicolon to terminate the variable declaration. It does not find the semicolon on the current line, so it reads on and finds the 'begin' keyword at the start of the next line. At this point it finally knows something is wrong... }

```
program Produce;
var
  I: Integer
begin               { <-- Error message here: ';' expected but 'BEGIN' found
  }
end.
```

{ In this case, just the semicolon was missing - a frequent case in practice. In general, have a close look at the line where the error message appears, and the line above it to find out whether something is missing or misspelled. }

```
program Solve;
var
  I: Integer;       { Semicolon was missing }
begin
end.
```

"Identifier redeclared: '<name>'"

Examples Compiler error messages

Description

The given identifier has already been declared in this scope - you are trying to reuse its name for something else.

Examples

{ Here the name of the program is the same as that of the variable - we need to change one of them to make the compiler happy. }

```
program Tests;
var
  Tests: Integer;
begin
end.
```

```
program Tests;
var
  TestCnt: Integer;
begin
end.
```

"Duplicate case label"

Examples Compiler error messages

Description

This error message occurs when there is more than one case label with a given value in a case statement.

Examples

{ Here we did not pay attention and mentioned the case label 0 twice. }

```
program Produce;

function DigitCount(I: Integer): Integer;
begin
  case Abs(I) of
  0:                    DigitCount := 1;
  0        ..9:         DigitCount := 1;   { <-- Error message here }
  10       ..99:        DigitCount := 2;
  100      ..999:       DigitCount := 3;
  1000     ..9999:      DigitCount := 4;
  10000    ..99999:     DigitCount := 5;
  100000   ..999999:    DigitCount := 6;
  1000000  ..9999999:   DigitCount := 7;
  10000000 ..99999999:  DigitCount := 8;
  100000000..999999999: DigitCount := 9;
  else                  DigitCount := 10;
  end;
end;

begin
  Writeln( DigitCount(12345) );
end.
```

{ In general, the problem might not be so easy to spot when you have symbolic constants and ranges of case labels - you might have to write down the real values of the constants to find out what is wrong. }

```
program Solve;

function DigitCount(I: Integer): Integer;
begin
  case Abs(I) of
  0        ..9:         DigitCount := 1;
  10       ..99:        DigitCount := 2;
  100      ..999:       DigitCount := 3;
  1000     ..9999:      DigitCount := 4;
  10000    ..99999:     DigitCount := 5;
  100000   ..999999:    DigitCount := 6;
  1000000  ..9999999:   DigitCount := 7;
  10000000 ..99999999:  DigitCount := 8;
  100000000..999999999: DigitCount := 9;
  else                  DigitCount := 10;
  end;
end;

begin
  Writeln( DigitCount(12345) );
end.
```

"Label expected"

Examples Compiler error messages

Description

This error message occurs if the identifier given in a **goto** statement or used as a label in inline assembly is not declared as a label.

Examples

```
program Produce;

begin
  if 2*2 <> 4 then
    goto Exit; { <-- Error message here: Exit is also a standard procedure }
  { ... }
Exit:              { Additional error messages here }
end.

program Solve;
label
  Exit;            { Labels must be declared in Pascal }
begin
  if 2*2 <> 4 then
    goto Exit;
  { ... }
Exit:
end.
```

"For loop control variable must be simple local variable"

Examples Compiler error messages

Description

This error message is given when the control variable of a **for** statement is not a simple variable (but a component of a record, for instance), or if it is not local to the procedure containing the for statement. For backward compatibility reasons, it is legal to use a global variable as the control variable - the compiler gives a warning in this case. Note that using a local variable will also generate more efficient code.

Examples

```
program Produce;

var
  I: Integer;
  A: array[0..9] of Integer;

procedure Init;
begin
  for I := Low(A) to High(a) do  { <-- Warning given here }
    A[I] := 0;
end;

begin
  Init;
end.
```

```
program Solve;
var
  A: array[0..9] of Integer;

procedure Init;
var
  I: Integer;
begin
  for I := Low(A) to High(a) do
    A[I] := 0;
end;

begin
  Init;
end.
```

"For loop control variable must have ordinal type"

Examples Compiler error messages

Description

The control variable of a **for** loop must have type Boolean, Char, WideChar, Integer, an enumerated type, or a subrange type.

Examples

{ The example uses a variable of type Real as the for loop control variable, which is illegal. }

```
program Produce;
var
  X: Real;
begin { Plot sine wave }
  for X := 0 to 2*Pi/0.2 do                          { <-- Error message
  here }
    Writeln( '*': Round((Sin(X*0.2) + 1)*20) + 1 );
end.
```

{ The most obvious ordinal type to use here is Integer, which works just fine. }

```
program Solve;
var
  X: Integer;
begin { Plot sine wave }
  for X := 0 to Round(2*Pi/0.2) do
    Writeln( '*': Round((Sin(X*0.2) + 1)*20) + 1 );
end.
```

"Types of actual and formal var parameters must be identical"

Examples Compiler error messages

Description

For a variable parameter, the actual argument must be of the exact type of the formal parameter.

Examples

{ Arguments C1 and C2 are not acceptable to SwapBytes, although they have the exact memory representation and range that a Byte has. }

```
program Produce;

procedure SwapBytes(var B1, B2: Byte);
var
  Temp: Byte;
begin
  Temp := B1; B1 := B2; B2 := Temp;
end;

var
  C1, C2: 0..255;     { Similar to a byte, but NOT identical }
begin
  SwapBytes(C1,C2);   { <-- Error message here }
end.
```

{ So you actually have to declare C1 and C2 as Bytes to make this example compile. }

```
program Solve;

procedure SwapBytes(var B1, B2: Byte);
var
  Temp: Byte;
begin
  Temp := B1; B1 := B2; B2 := Temp;
end;

var
  C1, C2: Byte;
begin
  SwapBytes(C1,C2);   { <-- Error message here }
end.
```

"Too many actual parameters"

Examples Compiler error messages

Description

This error message occurs when a procedure or function call gives more parameters than the procedure or function declaration specifies.
Additionally, this error message occurs when an OLE automation call has too many (more than 255), or too many named parameters.

Examples

{ It would have been convenient for Max to accept three parameters... }

```
program Produce;

function Max(A,B: Integer): Integer;
begin
  if A > B then Max := A else Max := B
end;

begin
  Writeln( Max(1,2,3) );   { <-- Error message here }
end.
```

{ Normally, you would change to call site to supply the right number of parameters. Here, we have chosen to show you how to implement Max with an unlimited number of arguments. Note that now you have to call it in a sligtly different way. }

```
program Solve;

function Max(const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
      Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );
end.
```

"Not enough actual parameters"

Examples Compiler error messages

Description

This error message occurs when a call to procedure or function gives fewer parameters than specified in the procedure or function declaration. This can also occur for calls to standard procedures or functions.

Examples

{ The standard procedure Val has one additional parameter to return an error code in. The example did not supply that parameter. }

```
program Produce;
var
  X: Real;
begin
  Val('3.141592', X);   { <-- Error message here }
end.
```

{ Typically, you will check the call against the declaration of the procedure called or the help, and you will find you forgot about a parameter you need to supply. }

```
program Solve;
var
  X: Real;
  Code: Integer;
begin
  Val('3.141592', X, Code);
end.
```

"Variable required"

Examples Compiler error messages

Description

This error message occurs when you try to take the address of an expression or a constant.

Examples

{ A constant like 1 does not have a memory address, so you cannot apply the @ operator or the Addr standard function to it. }

```
program Produce;
var
  I: Integer;
  PI: ^Integer;
begin
  PI := Addr(1);
end.
```

{ You need to make sure you take the address of variable. }

```
program Solve;
var
  I: Integer;
  PI: ^Integer;
begin
  PI := Addr(I);
end.
```

"Declaration of <name> differs from previous declaration"

Examples Compiler error messages

Description

This error message occurs when the declaration of a procedure, function, method, constructor, or destructor differs from its previous (forward) declaration.
This error message also occurs when you try to override a virtual method, but the overriding method has a different parameter list, calling convention etc.

Examples

{ As you can see, there are a number of reasons for this error message to be issued. }

```
program Produce;

type
  MyClass = class
    procedure Proc(Inx: Integer);
    function Func: Integer;
    procedure Load(const Name: string);
    procedure Perform(Flag: Boolean);
    constructor Create;
    destructor Destroy(Msg: string); override;      { <-- Error message here
  }
    class function NewInstance: MyClass; override;  { <-- Error message here
  }
  end;

procedure MyClass.Proc(Index: Integer);             { <-- Error message here
  }
begin
end;

function MyClass.Func: Longint;                     { <-- Error message here
  }
begin
end;

procedure MyClass.Load(Name: string);               { <-- Error message here
  }
begin
end;

procedure MyClass.Perform(Flag: Boolean); cdecl;    { <-- Error message here
  }
begin
end;

procedure MyClass.Create;                           { <-- Error message here
  }
begin
end;

function MyClass.NewInstance: MyClass;              { <-- Error message here
  }
begin
end;

begin
end.
```

{ You need to carefully compare the 'previous declaration' with the one that causes the error to determine what is different between the two. }

```
program Solve;

type
  MyClass = class
```

```
    procedure Proc(Inx: Integer);
    function Func: Integer;
    procedure Load(const Name: string);
    procedure Perform(Flag: Boolean);
    constructor Create;
    destructor Destroy; override;                    { No parameters }
    class function NewInstance: TObject; override;   { Result type    }
  end;

procedure MyClass.Proc(Inx: Integer);                { Parameter name   }
begin
end;

function MyClass.Func: Integer;                      { Result type   }
begin
end;

procedure MyClass.Load(const Name: string);          { Parameter kind   }
begin
end;

procedure MyClass.Perform(Flag: Boolean);            { Calling convention }
begin
end;

constructor MyClass.Create;                          { constructor }
begin
end;

class function MyClass.NewInstance: TObject;         { class function }
begin
end;

begin
end.
```

"Illegal character in input file: '<Char>' (<Hexadecimal value>)"

Examples Compiler error messages

Description

The compiler found a character that is illegal in Pascal programs. This error message is caused most often by errors with string constants or comments.

Examples

{ Here a programmer fell back to C++ habits and quoted a string with double quotes. }

```
program Produce;

begin
  Writeln("Hello world!");    { <-- Error messages here }
end.
```

{ The solution is to use single quotes. In general, you need to delete the illegal character. }

```
program Solve;

begin
  Writeln('Hello world!');    { Need single quotes in Pascal }
end.
```

"'<name>' is not a type identifier"

Examples Compiler error messages

Description

This error message occurs when the compiler expected the name of a type, but the name it found did not stand for a type.

Examples

{ The example erroneously uses the name of the variable, not the name of the type, as the type of the argument. }

```
program Produce;
type
  TMyClass = class
    Field: Integer;
  end;
var
  MyClass: TMyClass;

procedure Proc(C: MyClass);   { <-- Error message here }
begin
end;

begin
end.
```

{ Make sure the offending identifier is indeed a type - maybe it was misspelled, or another identifier of the same name hides the one you meant to refer to. }

```
program Solve;
type
  TMyClass = class
    Field: Integer;
  end;
var
  MyClass: TMyClass;

procedure Proc(C: TMyClass);
begin
end;

begin
end.
```

"File not found: <Filename>"

Examples Compiler error messages

Description

This error message occurs when the compiler cannot find an input file. This can be a source file, a compiled unit file (.dcu file), an include, an object file or a resource file. Check the spelling of the name and the relevant search path.

Examples

```
program Produce;
uses SysUtis;       { <-- Error message here }
begin
end.

program Solve;
uses SysUtils;      { Fixed typo }
begin
end.
```

"Could not create output file <Filename>"

Compiler error messages

Description

The compiler could not create an output file. This can be a compiled unit file (.dcu file), an executable file, a map file or an object file. Most likely causes are a nonexistent directory or a write protected file or disk.

"Seek error on <Filename>"

Compiler error messages

Description

The compiler encountered a seek error on an input or output file. This should never happen - if it does, the most likely cause is corrupt data.

"Read error on <Filename>"

Compiler error messages

Description

The compiler encountered a read error on an input file. This should never happen - if it does, the most likely cause is corrupt data.

"Write error on <Filename>"

Compiler error messages

Description

The compiler encountered a write error while writing to an output file. Most likely, the output disk is full.

"Close error on <Filename>"

Compiler error messages

Description

The compiler encountered an error while closing an input or output file. This should rarely happen. If it does, the most likely cause is a full or bad disk.

"Bad file format: <Filename>"

Compiler error messages

Description

This error occurs if an object file loaded with a **$L** or **$LINK** directive is not of the correct format. Several restrictions must be met:

- Check the naming restrictions on segment names in the help file
- Not more than 10 segments
- Not more than 255 external symbols
- Not more than 50 local names in LNAMES records
- LEDATA and LIDATA records must be in offset order
- No THREAD subrecords are supported in FIXU32 records
- Only 32-bit offsets can be fixed up
- Only segment and self relative fixups
- Target of a fixup must be a segment, a group or an EXTDEF
- Object must be 32-bit object file
- Various internal consistency condition that should only fail if the object file is ccorrupted.

"Out of memory"

Compiler error messages

Description

The compiler ran out of memory. This should rarely happen. If it does, make sure your swap file is large enough and that there is still room on the disk.

"Circular unit reference to <Unitname>"

Examples Compiler error messages

Description

One or more units use each other in their interface parts. As the compiler has to translate the interface part of a unit before any other unit can use it, the compiler must be able to find a compilation order for the interface parts of the units. Check whether all the units in the uses clauses are really necessary, and whether some can be moved to the implementation part of a unit instead.

Examples

{ The problem is caused because A and B use each other in their interface sections. }

```
unit A;
interface
uses B;            { A uses B, and B uses A }
implementation
end.

unit B;
interface
uses A;
implementation
end.
```

{ You can break the cycle by moving one or more uses to the implementation part. }

```
unit A;
interface
uses B;          { Compilation order: B.interface, A, B.implementation }
implementation
end.

unit B;
interface
implementation
uses A;          { Moved to the implementation part }
end.
```

"Bad unit format: <Filename>"

Compiler error messages

Description

This error occurs if a compiled unit file (.dcu file) has a bad format. Most likely, the .dcu file has been corrupted. Recompile the file or reinstall Delphi.

"PACKED not allowed here"

Examples Compiler error messages

Description

The **packed** keyword is only legal for set, array, record, object, class and file types. In contrast to the 16-bit version of Delphi, **packed** will affect the layout of record, object and class types.

Examples

{ Packed can not be applied to a real type - if you want to conserve storage, you need to use the smallest real type, type Single. }

```
program Produce;
type
  SmallReal = packed Real;
begin
end.

program Solve;
type
  SmallReal = Single;
begin
end.
```

"Label declaration not allowed in interface part"

Examples Compiler error messages

Description

This error occurs when you declare a label in the interface part of a unit.

Examples

{ It is just illegal to declare a label in the interface section of a unit. }

```
unit Produce;
interface
label 99;
implementation
begin
99:
end.
```

{ You have to move it to the implementation section. }

```
unit Solve;
interface
implementation
label 99;
begin
99:
end.
```

"Statements not allowed in interface part"

Examples Compiler error messages

Description

The interface part of a unit can only contain declarations, not statements. Move the bodies of procedures to the implementation part.

Examples

{ We got carried away and gave MyProc a body right in the interface section. }

```
unit Produce;

interface

procedure MyProc;
begin                 { <-- Error message here }
end;

implementation

begin
end.
```

{ We need move the body to the implementation section - then it's fine. }

```
unit Solve;

interface

procedure MyProc;

implementation

procedure MyProc;
begin
end;

begin
end.
```

"Unit1> was compiled with a different version of <Unit2>"

Compiler error messages

Description

This error occurs when the declaration of a symbol declared in the interface part of a unit has changed, and the compiler cannot recompile a unit that relies on this declaration because the source is not available to it.
There are several possible solutions - recompile Unit1 (assuming you have the source code available), use an older version of Unit2 or change Unit2, or get a new version of Unit1 from whoever has the source code to it.

"Unterminated string"

Examples Compiler error messages

Description

The compiler did not find a closing apostrophe at the end of a character string.
Note that character strings cannot be continued onto the next line - however, you can use the + operator to concatenate two character strings on separate lines.

Examples

{ We just forgot the closing quote at the string - no big deal, happens all the time. }

```
program Produce;

begin
  Writeln('Hello world!);   { <-- Error message here - }
end.
```

{ So we supplied the closing quote, and the compiler is happy. }

```
program Solve;

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

"Syntax error in real number"

Examples Compiler error messages

Description

This error message occurs if the compiler finds the beginning of a scale factor (an 'E' or 'e' character) in a number, but no digits follow it.

Examples

{ In the example, we put a space after '3.0E' - now for the compiler the number ends here, and it is incomplete. }

```
program Produce;
const
  SpeedOfLight = 3.0E 8;     { <-- Error message here }
begin
end.
```

{ We could have just deleted the blank, but we put in a '+' sign because it looks a little nicer. }

```
program Solve;
const
  SpeedOfLight = 3.0E+8;
begin
end.
```

"Procedure too long: exceeds 32K"

Compiler error messages

Description

This error message is unused on the Intel processor.

"Illegal type in Write/Writeln statement"

Examples Compiler error messages

Description

This error occurs when you try to output a type in a Write or Writeln statement that is not legal.

Examples

{ It would have been convenient to use a writeln statement to output Color, wouldn't it? }

```
program Produce;
type
  TColor = (red,green,blue);
var
  Color: TColor;
begin
  Writeln(Color);
end.
```

{ Unfortunately, that is not legal, and we have to do it with an auxiliary array. }

```
program Solve;
type
  TColor = (red,green,blue);
var
  Color: TColor;
const
  ColorString: array[TColor] of string = ('red', 'green', 'blue');
begin
  Writeln(ColorString[Color]);
end.
```

"Illegal type in Read/Readln statement"

Examples Compiler error messages

Description

This error occurs when you try to read a variable in a Read or Readln that is not of a legal type. Check the type of the variable and make sure you are not missing a dereferencing, indexing or field selection operator.

Examples

{ We cannot read variables of enumerated types directly. }

```
program Produce;
type
  TColor = (red,green,blue);
var
  Color: TColor;
begin
  Readln(Color);     { <-- Error message here }
end.
```

{ The solution is to read a string, and look up that string in an auxiliary table. In the example above, we didn't bother to do error checking - any string will be treated as 'blue'. In practice, we would probably output an error message and ask the user to try again. }

```
program Solve;
type
  TColor = (red,green,blue);
var
  Color: TColor;
  InputString: string;
const
  ColorString: array[TColor] of string = ('red', 'green', 'blue');
begin
  Readln(InputString);
  Color := red;
  while (color < blue) and (ColorString[color] <> InputString) do
    Inc(color);
end.
```

"Strings may have at most 255 elements"

Examples Compiler error messages

Description

This error message occurs when you declare a short string type with more than 255 elements, if you assign a string literal of more than 255 characters to a variable of type ShortString, or when you have more than 255 characters in a single character string.

Note that you can construct long string literals spanning more than one line by using the '+' operator to concatenate several string literals.

Examples

{ In the example above, the length of the string is just one beyond the limit. }

```
program Produce;
var
  LongString: string[256];  { <-- Error message here }
begin
end.
```

{ The most convenient solution is to use the new long strings - then you don't even have to spend any time thinking about what a reasonable maximum length would be. }

```
program Solve;
var
  LongString: AnsiString;
begin
end.
```

"Unexpected end of file in comment started on line <Number>"

Examples Compiler error messages

Description

This error occurs when you open a comment, but do not close it. Note that a comment started with '{' must be closed with '}', and a comment started with '(*' must be closed with '*)'.

Examples

{ So the example just didn't close the comment. }

```
program Produce;
{ Let's start a comment here but forget to close it
begin
end.
```

{ Doing so fixes the problem. }

```
program Solve;
{ Let's start a comment here and not forget to close it }
begin
end.
```

"Constant or type identifier expected"

Examples Compiler error messages

Description

This error message occurs when the compiler expects a type, but finds a symbol that is neither a constant (a constant could start a subrange type), nor a type identifier.

Examples

{ Here, ExceptionClass is a variable, not a type. }

```
program Produce;
var
  C: ExceptionClass; { ExceptionClass is a variable in System }
begin
end.
```

{ You need to make sure you specify a type. Maybe the identifier is misspelled, or it is hidden by some other identifier, for example from another unit. }

```
program Solve;
var
  C: Exception; { Exception is a type in SysUtils }
begin
end.
```

"Invalid compiler directive: '<Directive>'"

Examples Compiler error messages

Description

This error message means there is an error in a compiler directive or in a command line option.
Here are some possible error situations:

- An external declaration was syntactically incorrect.
- A command line option or an option in a DCC32.CFG file was not recognized by the compiler or was invalid. For example, '-$M100' is invalid because the minimum stack size must be at least 1024.
- The compiler found a $XXXXX directive, but could not recognize it. It was probably misspelled.
- The compiler found a $ELSE or $ENDIF directive, but no preceding $IFDEF, $IFNDEF or $IFOPT directive.
- {$IFOPT} was not followed by a a switch option and a + or -.
- The long form of a switch directive was not followed by ON or OFF.
- A directive taking a numeric parameter was not followed by a valid number.
- The $DESCRIPTION directive was not followed by a string.
- The $APPTYPE directive was not followed by CONSOLE or GUI.
- The $ENUMSIZE directive (short form $Z) was not followed by 1,2 or 4.

Examples

{ The example shows three typical error situations, and the last two errors are caused by the compiler not having recognized $If. }

```
{$Description Copyright Borland International 1996}    { <-- Error here }
program Produce;
{$ApType Console}                                       { <-- Error here }

begin
{$If O+}                                               { <-- Error here }
  Writeln('Optimizations are ON');
{$Else}                                                { <-- Error here }
  Writeln('Optimizations are OFF');
{$Endif}                                               { <-- Error here }
  Writeln('Hello world!');
end.
```

{ So $Description needs a quoted string, we need to spell $AppType right, and checking options is done with $IfOpt. With these changes, the example compiles fine. }

```
{$Description 'Copyright Borland International 1996'}  { Need string }
program Solve;
{$AppType Console}                                      { AppType }

begin
{$IfOpt O+}                                            { IfOpt }
  Writeln('Optimizations are ON');
{$Else}                                                { Now fine }
  Writeln('Optimizations are OFF');
{$Endif}                                               { Now fine }
  Writeln('Hello world!');
end.
```

"Bad global symbol definition: '<name>' in object file '<Filename>'"

Compiler error messages

Description

This warning is given when an object file linked in with a **$L** or **$LINK** directive contains a definition for a symbol that was not declared in Pascal as an **external** procedure, but as something else (such as a variable). The definition in the object will be ignored in this case.

"Invalid relocation information"

Compiler error messages

Description

This error message is currently unused.

"Class or object types only allowed in type section"

Examples Compiler error messages

Description

Class or object types must always be declared with an explicit type declaration in a type section - unlike record types, they cannot be anonymous. The main reason for this is that there would be no way you could declare the methods of that type - after all, there is no type name.

Examples

{ The example tries to declare a class type within a variable declaration - that is not legal. }

```
program Produce;

var
  MyClass: class
    Field: Integer;
  end;

begin
end.
```

{ The solution simply is to introduce a type declaration for the class type. Alternatively, you could have changed the class type to a record type. }

```
program Solve;

type
  TMyClass = class
    Field: Integer;
  end;

var
  MyClass: TMyClass;

begin
end.
```

"Local class or object types not allowed"

Examples Compiler error messages

Description

Class and object cannot be declared local to a procedure.

Examples

{ MyProc tries to declare a class type locally, which is illegal. }

```
program Produce;

procedure MyProc;
type
  TMyClass = class
    Field: Integer;
  end;
begin
{ ... }
end;

begin
end.
```

{ The solution is to simply move out the declaration of the class or object type to the global scope. }

```
program Solve;

type
  TMyClass = class
    Field: Integer;
  end;

procedure MyProc;
begin
{ ... }
end;

begin
end.
```

"Virtual constructors are not allowed"

Examples Compiler error messages

Description

Unlike class types, object types can only have static constructors.

Examples

{ The example tries to declare a virtual constructor, which does not really make sense for object types and is therefore illegal. }

```
program Produce;

type
  TMyObject = object
    constructor Init; virtual;
  end;

constructor TMyObject.Init;
begin
end;

begin
end.
```

{ The solution is to either make the constructor static, or to use a new-style class type which can have a virtual constructor }

```
program Solve;

type
  TMyObject = object
    constructor Init;
  end;

constructor TMyObject.Init;
begin
end;

begin
end.
```

"Could not compile used unit '<Unitname>'"

Compiler error messages

Description

This fatal error is given when a unit used by another could not be compiled. In this case, the compiler gives up compilation of the dependent unit because it is likely very many errors will be encountered as a consequence.

"Left side cannot be assigned to"

Examples Compiler error messages

Description

This error message is given when you try to modify a read-only object like a constant, a constant parameter, or the return value of a function.

Examples

{ The example assigns to constant parameter, to a constant, and to the result of a function call. All of these are illegal. }

```
program Produce;

const
  C = 1;

procedure P(const S: string);
begin
  S := 'changed';            { <-- Error message here }
end;

function F: PChar;
begin
  F := 'Hello';              { This is fine - we are setting the return
  value }
end;

begin
  C := 2;                    { <-- Error message here }
  F := 'h';                  { <-- Error message here }
end.
```

{ There two ways you can solve this kind of problem: either you change the definition of whatever you are assigning to, so the assignment becomes legal, or you eliminate the assignment. }

```
program Solve;

var
  C: Integer = 1;           { Use an initialized variable }

procedure P(var S: string);
begin
  S := 'changed';            { Use variable parameter }
end;

function F: PChar;
begin
  F := 'Hello';              { This is fine - we are setting the return
  value }
end;

begin
  C := 2;
  F^ := 'h';                 { This compiles, but will crash at runtime }
end.
```

"Unsatisfied forward or external declaration: '<Procedurename>'"

Examples Compiler error messages

Description

This error message appears when you have a forward or external declaration of a procedure or function, or a declaration of a method in a class or object type, and you don't define the procedure, function or method anywhere. Maybe the definition is really missing, or maybe its name is just misspelled.

Note that a declaration of a procedure or function in the interface section of a unit is equivalent to a forward declaration - you have to supply the implementation (the body of the procedure or function) in the implementation section. Similarly, the declaration of a method in a class or object type is equivalent to a forward declaration.

Examples

{ The definition of Sum has an easy to spot typo - it may not be so obvious in a real world example with a couple thousand lines between the forward declaration and the definition of a procedure. }

```
program Produce;

type
  TMyClass = class
    constructor Create;
  end;

function Sum(const A: array of Double): Double; forward;

function Summ(const A: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I:= 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

{ So you just need to make sure the definitions of your procedures, functions and methods are all there, and spelled correctly. }

```
program Solve;

type
  TMyClass = class
    constructor Create;
  end;

constructor TMyClass.Create;
begin
end;

function Sum(const A: array of Double): Double; forward;

function Sum(const A: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I:= 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

"Missing operator or semicolon"

See also Examples Compiler error messages

Description

This error message appears if there is no operator between two subexpressions, or no semicolon between two statements. Often, a semicolon is missing on the previous line.

Examples

{ The first statement in the example has two errors - a '+' operator and a semicolon are missing. The first error is reported on this statement, the second on the following line. }

```
program Produce;
var
  I: Integer;
begin
  I := 1 2                { <-- Error message here }
  if I = 3 then           { <-- Error message here }
    Writeln('Fine')
end.
```

{ The solution is to make sure the necessary operators and semicolons are there. }

```
program Solve;
var
  I: Integer;
begin
  I := 1 + 2;             { We were missing a '+' operator and a
  semicolon }
  if I = 3 then
    Writeln('Fine')
end.
```

See also
Compound statements

"Incompatible types"

Examples Compiler error messages

Description

This error message occurs when the compiler expected two types to be compatible (meaning very similar), but in fact, they turned out to be different. This error occurs in many different situations - for example when a **read** or **write** clause in a property mentions a method whose parameter list does not match the property, or when a parameter to a standard procedure or function is of the wrong type.

Examples

{ The standard function Hi expects an argument of type Integer or Word, but we supplied an array instead. }

```
program Produce;
var
  A: array[0..9] of Char;
  I: Integer;
begin
  I:= Hi(A);
end.
```

{ We really meant to use the standard function High, not Hi. }

```
program Solve;
var
  A: array[0..9] of Char;
  I: Integer;
begin
  I:= High(A);
end.
```

"Missing parameter type"

Examples Compiler error messages

Description

This error message is issued when a parameter list gives no type for a value parameter. Leaving off the type is legal for constant and variable parameters.

Examples

{ We intended procedure P to have two integer parameters, but we put a semicolon instead of a comma after the first parameters. The function ComputeHash was supposed to have an untyped first parameter, but untyped parameters must be either variable or constant parameters - they cannot be value parameters. }

```
program Produce;

procedure P(I;J: Integer);                              { <-- Error message
  here }
begin
end;

function ComputeHash(Buffer; Size: Integer): Integer; { <-- Error message
  here }
begin
end;

begin
end.
```

{ The solution in this case was to fix the type in P's parameter list, and to declare the Buffer parameter to ComputeHash as a constant parameter, because we don't intend to modify it. }

```
program Solve;

procedure P(I,J: Integer);
begin
end;

function ComputeHash(const Buffer; Size: Integer): Integer;
begin
end;

begin
end.
```

"Illegal reference to symbol '<name>' in object file '<Filename>'"

Compiler error messages

Description

This error message is given if an object file loaded with a **$L** or **$LINK** directive contains a reference to a Pascal symbol that is not a procedure, function, variable, typed constant or thread local variable.

"Line too long (more than 255 characters)"

Compiler error messages

Description

This error message is given when the length of a line in the source file exceeds 255 characters. Usually, you can divide the long line into two shorter lines.
If you need a really long string constant, you can break it into several pieces on consecutive lines that you concatenate with the '+' operator.

"Unknown directive: '<Directive>'"

Examples Compiler error messages

Description

This error message appears when the compiler encounters an unknown directive in a procedure or function declaration. The directive is probably misspelled, or a semicolon is missing.

Examples

{ In the declaration of P, the calling convention is misspelled. In the declaration of Q and GetLastError, we're missing a semicolon. }

```
program Produce;

procedure P; stcall;
begin
end;

procedure Q forward;

function GetLastError: Integer external 'kernel32.dll';

begin
end.
```

{ The solution is to make sure the directives are spelled correctly, and that the necessary semicolons are there. }

```
program Solve;

procedure P; stdcall;
begin
end;

procedure Q; forward;

function GetLastError: Integer; external 'kernel32.dll';

begin
end.
```

"This type cannot be initialized"

Examples Compiler error messages

Description

File types (including type Text) and the type Variant cannot be initialized. That is, you cannot declare typed constants or initialized variables of these types.

Note For compatibility with older versions of Delphi and Borland Pascal, Object Pascal supports a compiler directive to allow assignments to typed constants. To enable writeable typed constants, set the $J directive to **{$J+}** or **{$WRITEABLECONSTS ON}**.

Examples

{ The example tries to declare an initialized variable of type Variant, which illegal. }

```
program Produce;

var
  V: Variant = 0;

begin
end.
```

{ The solution is simply to initialize a normal variable with an assignment statement. }

```
program Solve;

var
  V: Variant;

begin
  V := 0;
end.
```

"Number of elements differs from declaration"

Examples Compiler error messages

Description

This error message appears when you declare a typed constant or initialized variable of array type, but do not supply the appropriate number of elements.

Examples

{ The example declares an array of 10 elements, but the initialization only supplies 9 elements. }

```
program Produce;

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

begin
end.
```

{ We just had to supply the missing element to make the compiler happy. When initializing bigger arrays, it can be sometimes hard to see whether you have supplied the right number of elements. To help with that, you layout the source file in a way that makes counting easy (e.g. ten elements to a line), or you can put the index of an element in comments next to the element itself. }

```
program Solve;

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

begin
end.
```

"Label already defined: '<Labelname>'"

Examples Compiler error messages

Description

This error message is given when a label is set on more than one statement.

Examples

{ The example just tries to set label 1 twice. }

```
program Produce;
label 1;
begin
1:
  goto 1;
1:          { <-- Error message here }
end.
```

{ Make sure every label is set exactly once. }

```
program Solve;
label 1;
begin
1:
  goto 1;
end.
```

"Label declared and referenced, but not set: '<label>'"

Examples Compiler error messages

Description

You declared and used a label in your program, but the label definition was not encountered in the source code.

Examples

{ Label 10 is declared and used in the procedure 'Labeled', but the compiler never finds a defintion of the label. }

```
program Produce;

procedure Labeled;
label 10;
begin
  goto 10;
end;

begin
end.
```

{ The simple solution is to ensure that a declared and used label has a defintion, in the same scope, in your program. }

```
program Produce;

procedure Labeled;
label 10;
begin
  goto 10;
  10:
end;

begin
end.
```

"This form of method call only allowed in methods of derived types"

Examples Compiler error messages

Description

This error message is issued if you try to make a call to a method of an ancestor type, but you are in fact not in a method.

Examples

{ The example tries to call an inherited constructor in procedure Create, which is not a method. }

```
program Produce;

type
  TMyClass = class
    constructor Create;
  end;

procedure Create;
begin
  inherited Create;       { <-- Error message here }
end;

begin
end.
```

{ The solution is to make sure you are in fact in a method when using this form of call. }

```
program Solve;

type
  TMyClass = class
    constructor Create;
  end;

constructor TMyclass.Create;
begin
  inherited Create;
end;

begin
end.
```

"This form of method call only allowed for class methods"

Examples Compiler error messages

Description

You were trying to call a normal method by just supplying the class type, not an instance. This is only allowed for class methods and constructors, not normal methods and destructors.

Examples

{ The example tries to destroy the type TMyClass - this doesn't make sense and is therefore illegal. }

```
program Produce;

type
  TMyClass = class
    { ... }
  end;
var
  MyClass: TMyClass;

begin
  MyClass := TMyClass.Create;  { Fine, constructor }
  Writeln(TMyClass.ClassName); { Fine, class method }
  TMyClass.Destroy;            { <-- Error message here }
end.
```

{ Obviously, we really meant to destroy the instance of the type, not the type itself. }

```
program Solve;
type
  TMyClass = class
    { ... }
  end;
var
  MyClass: TMyClass;

begin
  MyClass := TMyClass.Create;  { Fine, constructor }
  Writeln(TMyClass.ClassName); { Fine, class method }
  MyClass.Destroy;             { Fine, called on instance }
end.
```

"Incompatible types: <text>"

See also Examples Compiler error messages

Description

The compiler has detected a difference between the declaration and use of a procedure.

Examples

{ The call of 'TakesParm0' will elicit an error because the type 'ProcedureParm0' expects a 'stdcall' procedure, whereas 'WrongConvention' is declared with the 'register' calling convention. Similarly, the call of 'TakesParm1' will fail because the parameter lists do not match. }

```
program Produce;

type
  ProcedureParm0 = procedure; stdcall;
  ProcedureParm1 = procedure(var X: Integer);

procedure WrongConvention; register;
begin
end;

procedure WrongParms(x, y, z: Integer);
begin
end;

procedure TakesParm0(p: ProcedureParm0);
begin
end;

procedure TakesParm1(p: ProcedureParm1);
begin
end;

begin
  TakesParm0(WrongConvention);
  TakesParm1(WrongParms);
end.
```

{ The solution to both of these problems is to simply ensure that the calling convention or the parameter lists matches the declaration. }

```
program Solve;

type
  ProcedureParm0 = procedure; stdcall;
  ProcedureParm1 = procedure(var X: Integer);

procedure RightConvention; stdcall;
begin
end;

procedure RightParms(var X: Integer);
begin
end;

procedure TakesParm0(p: ProcedureParm0);
begin
end;

procedure TakesParm1(p: ProcedureParm1);
begin
end;

begin
```

```
  TakesParm0(RightConvention);
  TakesParm1(RightParms);
end.
```

See also

Procedural types

Calling conventions

"Variable '<name>' might not have been initialized"

Examples Compiler error messages

Description

This warning is given if a variable has not been assigned a value on every code path leading to a point where it is used.

Examples

{ In an if statement, you have to make sure the variable is assigned in both branches. In a case statement, you need to add an else part to make sure the variable is assigned a value in every conceivable case. In a try-except construct, the compiler assumes that assignments in the try part may in fact not happen, even if they are at the very beginning of the try part and so simple that they cannot conceivably cause an exception. }

```
program Produce;
{$WARNINGS ON}
var
  B: Boolean;
  C: (Red,Green,Blue);

procedure Simple;
var
  I: Integer;
begin
  Writeln(I);        { <-- Warning here }
end;

procedure IfStatement;
var
  I: Integer;
begin
  if B then
    I := 42;
  Writeln(I);        { <-- Warning here }
end;

procedure CaseStatement;
var
  I: Integer;
begin
  case C of
    Red..Blue: I := 42;
  end;
  Writeln(I);        { <-- Warning here }
end;

procedure TryStatement;
var
  I: Integer;
begin
  try
    I := 42;
  except
    Writeln('Should not get here!');
  end;
  Writeln(I);        { <-- Warning here }
end;

begin
  B := False;
end.
```

{ The solution is to either add assignments to the code paths where they were missing, or to add an assignment before a conditional statement or a try-except construct. }

```
program Solve;
```

```
{$WARNINGS ON}
var
  B: Boolean;
  C: (Red,Green,Blue);

procedure Simple;
var
  I: Integer;
begin
  I := 42;
  Writeln(I);
end;

procedure IfStatement;
var
  I: Integer;
begin
  if B then
    I := 42
  else
    I := 0;
  Writeln(I);        { Need to assign I in the else part
end;

procedure CaseStatement;
var
  I: Integer;
begin
  case C of
    Red..Blue: I := 42;
  else       I := 0;
  end;
  Writeln(I);        { Need to assign I in the else part }
end;

procedure TryStatement;
var
  I: Integer;
begin
  I := 0;
  try
    I := 42;
  except
    Writeln('Should not get here!');
  end;
  Writeln(I);        { Need to assign I before the try }
end;

begin
  B := False;
end.
```

"Value assigned to '<name>' never used"

Examples Compiler error messages

Description

The compiler gives this hint message if the value assigned to a variable is not used. If optimization is enabled, the assignment is eliminated. This can happen because either the variable is not used anymore, or because it is reassigned before it is used.

This hint message does not indicate your program is wrong - it just means the compiler has determined there is an assignment that is not necessary. You can usually just delete this assignment - it will be dropped in the compiled code anyway if you compile with optimizations on. Sometimes, however, the real problem is that you assigned to the wrong variable, e.g. to meant to assign J but instead assigned I. So it is worthwhile to check the assignment in question carefully.

Examples

{ In procedure Propagate, the compiler is smart enough to realize that as variable I is not used after the while loop, it does not need to be incremented inside the while, and therefore the increment and the assignment before the while loop are also superfluous. In procedure TryFinally, the assignment to I before the try-finally construct is not necessary. If an exception happens, we don't execute the Writeln statement at the end, so the value of I does not matter. If no exception happens, the value of I seen by the Writeln statement is always 42. So the first assignment will not change the behaviour of the procedure, and can therefore be eliminated. }

```
program Produce;
{$HINTS ON}

procedure Simple;
var
  I: Integer;
begin
  I := 42;                { <-- Hint message here }
end;

procedure Propagate;
var
  I: Integer;
  K: Integer;
begin
  I := 0;                 { <-- Hint message here }
  Inc(I);                 { <-- Hint message here }
  K := 42;
  while K > 0 do
  begin
    if Odd(K) then
      Inc(I);             { <-- Hint message here }
    Dec(K);
  end;
end;

procedure TryFinally;
var
  I: Integer;
begin
  I := 0;                 { <-- Hint message here }
  try
    I := 42;
  finally
    Writeln('Reached finally');
  end;
  Writeln(I);             { Will always write 42 - if an exception happened,
                            we wouldn't get here }
end;

begin
end.
```

"Return value of function '<Functionname>' might be undefined"

Examples Compiler error messages

Description

This warning is given if the return value of a function has not been assigned a value on every code path. To put it a little differently, the function could execute in a way that never assigns anything to the return value.

Examples

{ The problem with procedure IfStatement and CaseStatement is that the result is not assigned in every code path. In TryStatement, the compiler assumes that an exception could happen before Result is assigned (in this case, the compiler is too conservative, obviously). }

```
program Produce;
{$WARNINGS ON}
var
  B: Boolean;
  C: (Red, Green, Blue);

function Simple: Integer;
begin
end;                            { <-- Warning here }

function IfStatement: Integer;
begin
  if B then
    Result := 42;
end;                            { <-- Warning here }

function CaseStatement: Integer;
begin
  case C of
    Red..Blue: Result := 42;
  end;
end;                            { <-- Warning here }

function TryStatement: Integer;
begin
  try
    Result := 42;
  except
    Writeln('Should not get here!');
  end;
end;                            { <-- Warning here }

begin
  B := False;
end.
```

{ The solution simply is to make sure there is an assignment to the result variable in every possible code path. }

```
program Solve;
{$WARNINGS ON}
var
  B: Boolean;
  C: (Red, Green, Blue);

function Simple: Integer;
begin
  Result := 42;
end;

function IfStatement: Integer;
begin
  if B then
    Result := 42
```

```
  else
    Result := 0;
end;

function CaseStatement: Integer;
begin
  case C of
    Red..Blue: Result := 42;
  else Result := 0;
  end;
end;

function TryStatement: Integer;
begin
  Result := 0;
  try
    Result := 42;
  except
    Writeln('Should not get here!');
  end;
end;

begin
  B := False;
end.
```

"Procedure FAIL only allowed in constructor"

Compiler error messages

Description

The standard procedure Fail can only be called from within a constructor - it is illegal otherwise.

"Procedure NEW needs constructor"

Examples Compiler error messages

Description

This error message is issued when an identifier given in the parameter list to New is not a constructor.

Examples

${ By mistake, we called New with the destructor, not the constructor. }

```
program Produce;

type
  PMyObject = ^TMyObject;
  TMyObject = object
    F: Integer;
    constructor Init;
    destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Done);               { <-- Error message here }
end.
```

{ Make sure you give the New standard function a constructor, or no additional argument at all. }

```
program Solve;

type
  PMyObject = ^TMyObject;
  TMyObject = object
    F: Integer;
    constructor Init;
    destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
end.
```

"Procedure DISPOSE needs destructor"

Examples Compiler error messages

Description

This error message is issued when an identifier given in the parameter list to Dispose is not a destructor.

Examples

{ In this example, we passed the constructor to Dispose by mistake. }

```
program Produce;

type
  PMyObject = ^TMyObject;
  TMyObject = object
    F: Integer;
    constructor Init;
    destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
  { ... }
  Dispose(P, Init);         { <-- Error message here }
end.
```

{ The solution is to either pass a destructor to Dispose, or to eliminate the second argument. }

```
program Solve;

type
  PMyObject = ^TMyObject;
  TMyObject = object
    F: Integer;
    constructor Init;
    destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
  Dispose(P, Done);
```

```
end.
```

"Assignment to FOR-Loop variable '<name>'"

Examples Compiler error messages

Description

It is illegal to assign a value to the for loop control variable inside the **for** loop. If the purpose is to leave the loop prematurely, use a Break or **goto** statement.

Examples

{ In this case, the programmer thought that assigning 99 to I would cause the program to exit the loop. }

```
program Produce;

var
  I: Integer;
  A: array[0..99] of Integer;
begin
  for I := 0 to 99 do
  begin
    if A[I] = 42 then
      I := 99;
  end;
end.
```

{ Using a break statement is a cleaner way to exit out of a for loop. }

```
program Solve;

var
  I: Integer;
  A: array[0..99] of Integer;
begin
  for I := 0 to 99 do
  begin
    if A[I] = 42 then
      Break;
  end;
end.
```

"FOR-Loop variable '<name>' may be undefined after loop"

Examples Compiler error messages

Description

This warning is issued if the value of a **for** loop control variable is used after the loop. You can only rely on the final value of a **for** loop control variable if the loop is left with a **goto** or Exit statement. The purpose of this restriction is to enable the compiler to generate efficient code for the **for** loop.

Examples

{ In the example, the Result variable is used implicitly after the loop, but it is undefined if we did not find the value - hence the warning. }

```
program Produce;
{$WARNINGS ON}

function Search(const A: array of Integer; Value: Integer): Integer;
begin
  for Result := 0 to High(A) do
    if A[Result] = Value then
      Break;
end;

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

begin
  Writeln( Search(A,11) );
end.
```

{ The solution simply is to assign the intended value to the control variable for the case where we don't exit the loop prematurely. }

```
program Solve;
{$WARNINGS ON}

function Search(const A: array of Integer; Value: Integer): Integer;
begin
  for Result := 0 to High(A) do
    if A[Result] = Value then
      Exit;
  Result := High(A)+1;
end;

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

begin
  Writeln( Search(A,11) );
end.
```

"TYPEOF can only be applied to object types with a VMT"

Examples Compiler error messages

Description

This error message is issued if you try to apply the standard function TypeOf to an object type that does not have a virtual method table. A simple workaround is to declare a dummy virtual method to force the compiler to generate a VMT.

Examples

{ The example tries to apply the TypeOf standard function to type TMyObject which does not have virtual functions, and therefore no virtual function table (VMT). }

```
program Produce;

type
  TMyObject = object
    procedure MyProc;
  end;

procedure TMyObject.MyProc;
begin
  { ... }
end;

var
  P: Pointer;
begin
  P := TypeOf(TMyObject);    { <-- Error message here }
end.
```

{ The solution simply is to introduce a dummy virtual function, or to eliminate the call to TypeOf. }

```
program Solve;

type
  TMyObject = object
    procedure MyProc;
    procedure Dummy; virtual;
  end;

procedure TMyObject.MyProc;
begin
  { ... }
end;

procedure TMyObject.Dummy;
begin
end;

var
  P: Pointer;
begin
  P := TypeOf(TMyObject);
end.
```

"Order of fields in record constant differs from declaration"

Examples Compiler error messages

Description

This error message occurs if record fields in a typed constant or initialized variable are not initialized in declaration order.

Examples

{ The example tries to initialize first Y, then X, in the opposite order from the declaration. }

```
program Produce;

type
  TPoint = record
    X, Y: Integer;
  end;

var
  Point: TPoint = (Y: 123; X: 456);

begin
end.
```

{ The solution simply is to adjust the order of initialization to correspond to the declaration order. }

```
program Solve;

type
  TPoint = record
    X, Y: Integer;
  end;

var
  Point: TPoint = (X: 456; Y: 123);

begin
end.
```

"Incompatible types: '<name>' and '<name>'"

Examples Compiler error messages

Description

This error message results when the compiler expected two types to be compatible (i.e. similar), but they turned out to be different.

Examples

{ Here a C++ programmer thought the division operator / would give him an integral result - not the case in Pascal. }

```
program Produce;

procedure Proc(I: Integer);
begin
end;

begin
  Proc( 22 / 7 ); { Result of / operator is Real }
end.
```

{ The solution in this case is to use the integral division operator div - in general, you have to look at your program very careful to decide how to resolve type incompatibilities. }

```
program Solve;

procedure Proc(I: Integer);
begin
end;

begin
  Proc( 22 div 7 ); { The div operator gives result type Integer }
end.
```

"Internal error: <ErrorCode>"

Compiler error messages

Description

You should never get this error message - it means there is a programming error in the compiler. If you do, please call Borland Tech Support and let us know the ErrorCode (e.g. "C1196") that appears in the error message. This will give us a rough indication what went wrong. It is even more helpful if you can give us an example program that produces this message.

"Unit name mismatch: '<Unitname>'"

Examples Compiler error messages

Description

The unit identifier does not match the unit file name. This can happen when mixing long file names and shorter unit identifiers.

Examples

{ In this case, the problem is that the compiler found the wrong unit, because the file names were truncated to 8 characters. }

----- Contents of MY_UNIT_.PAS ----------

```
unit My_Unit_With_A_Long_Name;
interface
implementation
end.
```

----- End of MY_UNIT_.PAS ----------

```
program Produce;
uses My_Unit_With_Another_Long_Name; { Will find MY_UNIT_.PAS if -P command
  line
               switch is active - but it's the wrong unit. }
begin
end.
```

{ The solution is to use long file names or to make sure the file names differ in the first 8 characters. Also, you need to make sure the file name of a unit corresponds to the unit name. }

"No identifiers referenced from unit <unit>"

Compiler error messages

Description

This error message is currently unused.

"Type '<name>' is not yet completely defined"

Examples Compiler error messages

Description

This error occurs if there is either a reference to a type that is just being defined, or if there is a forward declared class type in a type section and no final declaration of that type.

Examples

{ The example tries to refer to record type before it is completely defined. Also, because of a typo, the compiler never sees a complete declaration for TMyClass. }

```
program Produce;

type
  TListEntry = record
    Next: ^TListEntry;                    { <-- Error message here }
    Data: Integer;
  end;
  TMyClass = class;                       { <-- Error message here }
  TMyClassRef = class of TMyClass;
  TMyClasss = class                       { <-- Typo ... }
    { ... }
  end;

begin
end.
```

{ The solution for the first problem is to introduce a type declaration for an auxiliary pointer type. The second problem is fixed by spelling TMyClass correctly. }

```
program Solve;

type
  PListEntry = ^TListEntry;
  TListEntry = record
    Next: PListEntry;
    Data: Integer;
  end;
  TMyClass = class;
  TMyClassRef = class of TMyClass;
  TMyClass = class
    { ... }
  end;

begin
end.
```

"This Demo Version has been patched"

Compiler error messages

Description

This error message is currently unused.

"Integer constant or variable name expected"

Examples Compiler error messages

Description

This error message is issued if you try to declare an absolute variable, but the **absolute** directive is not followed by an integer constant or a variable name.

Examples

```
program Produce;

var
  I: Integer;
  J: Integer absolute Addr(I);   { <-- Error message here }

begin
end.
```

```
program Solve;

const
  Addr = 0;

var
  I: Integer;
  J: Integer absolute I;

begin
end.
```

"Invalid typecast"

See also Examples Compiler error messages

Description

This error message is issued for typecasts not allowed by the rules.
The following kinds of casts are allowed:

- Ordinal or pointer type to another ordinal or pointer type
- A character, string, array of character or pchar to a string
- An ordinal, real, string or variant to a variant
- A variant to an ordinal, real, string or variant
- A variable reference to any type of the same size

Note that casting real types to integer can be performed with the standard functions Trunc and Round. There are other transfer functions like Ord and Chr that might make your intention clearer.

Examples

{ This programmer thought he could cast a floating point constant to Integer, like in C. }

```
program Produce;

begin
  Writeln( Integer(Pi) );
end.
```

{ In Pascal, we have separate Transfer functions to convert floating point values to integer. }

```
program Solve;

begin
  Writeln( Trunc(Pi) );
end.
```

See also
Typecasting, Transfer routines, Trunc, Round, Chr, Ord

"User break - compilation aborted"

Compiler error messages

Description

This message is currently unused.

"Assignment to typed constant '<name>'"

Compiler error messages

Description

This warning message is currently unused.

"Segment/Offset pairs not supported in Borland 32-bit Pascal"

Examples Compiler error messages

Description

32-bit code no longer uses the segment/offset addressing scheme that 16-bit code used. In 16-bit versions of Borland Pascal, segment/offset pairs were used to declare absolute variables, and as arguments to the Ptr standard function. Note that absolute addresses should not be used in 32-bit protected mode programs. Instead, appropriate Win32 API functions should be called.

Examples

```
program Produce;

var
  VideoMode: Integer absolute $0040:$0049;

begin
  Writeln( Byte(Ptr($0040,$0049)^) );
end.

program Solve;
{ This version will compile, but will not run - absolute addresses are
  uncool }
var
  VideoMode: Integer absolute $0040*16+$0049;

begin
  Writeln( Byte(Ptr($0040*16+$0049)^) );
end.
```

The Delphi object model

Delphi introduces a number of changes in the definition of object types. A few of these changes are backward-compatible and can be incorporated into existing objects. Most, however, are exclusive to new-style objects.

You can declare old-style objects in one unit, new-style objects in another, and use both of them in the same program. Keep in mind, however, that there are certain differences between the objects, and you are responsible for handling those disparities.

Note that new-style object declarations use the reserved word **class**, while old-style objects still use **object**.

The following changes are compatible with objects using the version 7.0 and earlier object model:

- Public parts
- Private parts
- Protected parts
- Published parts
- Changes in object declarations
- Changes in object use
- Properties
- Changes in method dispatching

Changes in object declarations

See also

There are several important additions and changes in the way you declare objects and parts of objects. The most obvious change is that you use the reserved word **class**, rather than **object**, in the declaration. Using **object** declares an old-style object, while **class** declares a new-style object.

New-style object declarations differ in several other ways, however, including the following:

- Default ancestor
- Protected parts
- Published parts
- Class methods
- Forward class declaration

See also

Properties

Changes in method dispatching

Default ancestor

See also Example

The System unit defines an abstract object type called TObject that is the default ancestor of all newly declared objects. Because all objects now have a common ancestor, at a very basic level, you can treat them polymorphically.

Note You need not explicitly declare TObject as an ancestor.

TObject contains a rudimentary constructor and destructor, called Create and Destroy, respectively.

- TObject.Create is a constructor that allocates a dynamic instance of the object on the heap and initializes all its fields to zeros.
- TObject.Destroy disposes of the memory allocated by Create and destroys the object instance.

Example

The following type declarations are equivalent

```
type
  TMyObject = class
  ...
  end;

type
  TMyObject = class(TObject)
  ...
  end;
```

See also

Protected parts
Published parts
Class methods
Forward class declaration

Private parts

See also

Private parts restrict the visibility of a component identifier to the module containing the object type declaration. In other words, **private** component identifiers act like normal **public** component identifiers within the module that contains the object type declaration, but outside the module, any **private** component identifiers are unknown and inaccessible.

By placing related object types in the same module, these object types can gain access to each other's private components without making the private components known to other modules.

See also

Protected parts
Public parts
Published parts

Protected parts

See also

There **protected** directive operates like its counterparts, **private** and **public**, in that it is reserved only in the type declaration of a class.

The protected parts of a class and its ancestor classes can be accessed only through a class declared in the current unit.

Protection 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.

Protected methods are particularly useful in hiding the implementation of properties.

See also

Class methods
Default ancestor
Forward class declaration
Private parts
Public parts
Published parts

Published parts

See also

The **published** directive operates like its counterparts, **private**, **protected**, and **public**, in that it is reserved only in the type declaration of a class.

The visibility rules for published components are identical to those of public components. The only difference between published and public components is that run-time type information is generated for fields, methods, and properties that are declared in a **published** part. This run-time type information enables an application to dynamically query the fields, methods, and properties of an otherwise unknown object type.

Note: The Delphi Visual Class Library uses run-time type information to access the values of a component's properties when saving and loading form files. Furthermore, the Delphi IDE uses a component's run-time type information to determine the list of properties shown in the Object Inspector.

A object type cannot have **published** parts unless it is compiled in the {**$M+**} state or is derived from an object that was compiled in the {**$M+**} state. The **$M** compiler directive controls the generation of run-time type information for a class.

Fields defined in a **published** part must be of an object type. Fields of all other types are restricted to **public**, **protected**, and **private** parts.

Properties defined in a **published** part cannot be array properties. Furthermore, the type of a property defined in a **published** part must be an ordinal type, a real type (Single, Double, Extended, or Comp, but not Real), a string type, a small set type, an object type, or a method pointer type. A small set type is a set type with a base type whose lower and upper bounds have ordinal values between 0 and 15. In other words, a small set type is a set that fits in a byte or a word.

See also

Class methods
Default ancestor
Forward class declaration
Object types
Private parts
Protected parts
Public parts

Class methods

See also Example

Class methods are procedures and functions that operate on a class instead of an instance of the class. The implementation of the class method must not depend on the run-time values of any object fields.

To declare a class method, you put the reserved word **class** in front of the **procedure** or **function** keyword that starts the definition.

In the defining declaration of a class method, the identifier Self represents the class for which the method was activated. The type of Self in a class method is **class of** ClassType, where ClassType is the class type for which the method is implemented. Since Self does not represent an object reference in a class method, it is not possible to use Self to access fields, properties, and normal methods. It is however possible to call constructors and other class methods through Self.

A class method can be invoked through a class reference or an object reference. When invoked through an object reference, the class of the given object reference is passed as the Self parameter.

See also

Class type
Default ancestor
Protected parts
Published parts
Forward class declaration

Example

The following example declares the class method GetClassName and then accesses that method using a qualified method reference.

```
type
  TMyObject = class(TObject)
    class function GetClassName: string;
  end;

var
  MyObject: TMyObject;
  AString: string;
begin
  AString := TMyObject.GetClassName;
  MyObject := TMyObject.Create;
  AString := MyObject.GetClassName;
end;
```

Forward class declaration

See also Example

You can predeclare an object type to make it available to other object-type declarations without fully defining it. This is much like a forward declaration of a procedure or function, in that you declare the symbol, but fully define it later.

A forward declaration must be resolved by a normal declaration of the class within the same type declaration part. Forward declarations allow mutually dependent classes to be declared.

See also

Class methods
Default ancestor
Forward
Method implementations
Protected parts
Published parts

Example

To forward-declare a class, you do the following:

```
type
  TMyClass = class;
```

The class must then be fully defined within that same type-declaration block. A typical use looks something like this:

```
type
  TMyClass = class;
  TYourClass = class(TSomething)
    MyClassField: TMyClass;
    ...
  end;
  TMyClass = class(TObject)
    MyField: TMyType;
    ...
  end;
```

Changes in object use

See also

There are several changes in the way you use new-model objects.

- Reference model
- Method pointers
- Object references
- Run-time type information

See also

Changes in object declarations

Properties

Changes in method dispatching

Reference model

See also

All objects are dynamic instances (that is, allocated on the heap and accessed through a pointer), therefore, you do not need to declare separate object and pointer types and explicitly dereference object pointers.

Thus, in the old object model, you might have declared types as follows:

```
type
  PMyObject = ^TMyObject;
  TMyObject = object(TObject)
    MyField: PMyObject;
    constructor Init;
  end;
```

You would then construct an instance and perhaps access its field as follows:

```
var
  MyObject: PMyObject;
begin
  MyObject := New(PMyObject, Init);
  MyObject^.MyField := ...
end;
```

Delphi greatly simplifies this process by using a reference model that automatically assumes you want to dereference the pointer:

```
type
  TMyObject = class(TObject)
    MyField: TMyObject;
    constructor Create;
  end;
var
  MyObject: TMyObject;
begin
  MyObject := TMyObject.Create;
  MyObject.MyField := ...
end;
```

See also

Method pointers
Object references
Run-time type information

Method pointers

See also Example

Delphi allows you to declare procedural types that are object methods, enabling you to call particular methods of particular object instances at run time. The main advantage to method pointers is that they let you extend an object by delegation (that is, delegating some behavior to another object), rather than by deriving a new object and overriding methods.

Delphi uses method pointers to connect events with specific code in specific places, such as calling a method of a particular form when the user clicks a button. Thus, instead of deriving a new class from TButton and overriding its click behavior, the programmer connects the existing object to a method of a specific object instance (generally the form containing the button) that has the behavior you want.

The only difference between a method pointer declaration and a normal procedural type declaration is the use of the reserved words **of object** following the function or procedure prototype.

See also

Reference model
Object references
Run-time type information

Example

The following example declares a method pointer.

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
```

An object field of type TNotifyEvent is then assignment-compatible with any object method declared as procedure(Sender: TObject). For example,

```
type
  TAnObject = class(TObject)
    FOnClick: TNotifyEvent;
  end;
  TAnotherObject = class(TObject)
    procedure AMethod(Sender: TObject);
  end;
var
  AnObject: TAnObject;
  AnotherObject: TAnotherObject;
begin
  AnObject := TAnObject.Create;
  AnotherObject := TAnotherObject.Create;
  AnObject.FOnClick := AnotherObject.AMethod;
end;
```

Object references

See also Example

Delphi allows you to create pointers to object types, known as object references. This contrasts with object types, which allow operations to be performed on instances of objects. Objet reference types are sometimes referred to as metaclasses or metaclass types.

Using object references, you can do one of the following:

- Create instances of the type assigned to the reference
- Get information about the object type by calling class methods

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

To declare an object reference, you use the reserved word class. For example, to create a reference to the type TObject,

```
type
  TObjectRef = class of TObject;
```

Class type identifiers function as values of their corresponding object reference types.

A object reference type value is assignment-compatible with any ancestor object reference type. Therefore, during program execution, a object reference type variable can reference the object it was defined for or any descendant object defined from the object.

An object reference type variable may be **nil**, which indicates that the variable does not currently reference an object.

Every object 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;
```

Constructors and object references

A constructor can be invoked on a variable reference of a object reference type. This allows polymorphic construction of objects, that is, construction of objects whose actual type is not known at compile time.

Constructors that are invoked through object reference types are typically virtual. That way, the constructor implementation that ends up being called will depend on the actual (run-time) object type selected by the object reference.

See also

Constructors and destructors

Example

The following example shows legal uses of an object reference:

```
type
  TObjectRef = class of TObject;
  TDescendant = class(TObject)
  end;
var
  ObjectRef: TObjectRef;
  AnObject: TObject;
  ADescendant: TDescendant;
begin
  AnObject := TObject.Create;
  ADescendant := TDescendant.Create;
  ObjectRef := TObject;
  {insert code to demonstrate polymorphism }
  ObjectRef := TDescendant;
  {insert code to demonstrate polymorphism }
end;
```

Run-time type information

Delphi gives your programs access to object-type information at run time. Specifically, you can use a new operator, is, to determine whether a given object is of a given type or one of its descendants.

The expression

```
AnObject is TObjectType
```

evaluates as True if AnObject is assignment-compatible with objects of type TObjectType, meaning that AnObject is either of type TObjectType or of an object type descended from TObjectType.

You can also use run-time type information to assure safe typecasting of objects. The expression

```
AnObject as TObjectType
```

is equivalent to the typecast

```
TObjectType(AnObject)
```

except that using as raises an EInvalidCast exception if AnObject is not of a compatible type. The **as** expression is really equivalent to

```
if AnObject is TObjectType then TObjectType(AnObject)...
```

The **as** typecast is most useful as a shorthand when creating a **with..do** block that includes a typecast. For example, it is much simpler to write

```
with AnObject as TObjectType do ...
```

than the equivalent two-step process:

```
if AnObject is TObjectType then with TObjectType(AnObject) do ...
```

Properties

See also

Objects can have properties. Properties look to the end user like object fields, but internally they can encapsulate methods that read or write the value of the field. A property definition in an object declares a named attribute for objects of the class and the actions associated with reading and writing the attribute. Examples of properties are the caption of a form, the size of a font, the name of a database table, and so on.

Properties let you control access to protected fields or create side effects to changing what otherwise look like fields.

Properties are a natural extension of fields in an object. Both can be used to express attributes of an object, but whereas fields are merely storage locations which can be examined and modified at will, properties provide greater control over access to attributes, they provide a mechanism for associating actions with the reading and writing of attributes, and they allow attributes to be computed.

See the following topics for information about properties:

- Property syntax
- Fields for reading and writing properties
- Array properties
- Access methods
- Index specifiers
- Storage specifiers
- Property Overrides
- Read-only and write-only properties

See also

Changes in object declarations

Changes in object use

Changes in method dispatching

Property syntax

See also Example

The definition of a property specifies the name and type of the property, and the actions associated with reading (examining) and writing (modifying) the property. A property can be of any type except a file type.

-
-
-
-

Often you will store the value of the property in a private or protected field, then use the read and write methods to get and set the values. If the read and write methods are virtual, you can easily override them in descendant object types to change the reading and writing behavior of the property without affecting code that uses the property.

To set the value of a property,

- Use an assignment statement.

To retrieve the value of a property,

- Reference the property.

Example

The declarations below define an imaginary TCompass control which has a Heading property that can assume values from 0 to 359 degrees. The definition of the Heading property further states that its value is read from the FHeading field, and that its value is written using the SetHeading method.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    :
  end;
```

See also

Array properties
Fields for reading and writing properties
Index specifiers
Read-only and write-only properties
Storage specifiers

Fields for reading and writing properties

See also Example

When a property is referenced in an expression, its value is read using the field or method listed in the read specifier, and when a property is referenced in an assignment statement, its value is written using the field or method listed in the write specifier.

For example, the **read** part might return the value of a field, while the **write** part might set the field and produce other side effects.

Note: Unlike fields, properties cannot be passed as variable parameters, and it is not possible to take the address of a property using the @ operator. This is true even if the **read** and **write** specifiers both list a field identifier, thus ensuring that a future implementation of the property is free to change one or both access specifiers to list a method.

Example

The following example declares a property AValue and defines the its access method SetValue.

```
type
  TPropObject = class(TObject)
    FValue: Integer;
    procedure SetValue(NewValue: Integer);
    property AValue: Integer  read FValue write SetValue;
  end;
procedure TPropObject.SetValue(NewValue: Integer);
begin
  FValue := NewValue;
  UpdateScreen;
end;
```

See also

Property syntax

Array properties

Read-only and write-only properties

Array properties

See also Example

You can declare properties that look and act much like arrays, in that they have multiple values of the same type referred to by an index. Unlike an array, however, you cannot refer to the property as a whole, only to individual elements in the array. These properties are called array properties.

There are two important aspects to array properties:

- Declaring an array property
- Accessing an array property

Declaring an array property

The declaration of an array property is identical to the declaration of any other property, but you also declare an index parameter list which specifies the names and types of the indexes of the array property.

The format of an index parameter list is the same as a procedure's or function's formal parameter list, except that the parameter declarations are enclosed in square brackets instead of parentheses. Note that unlike array types, which can only specify ordinal type indexes, array properties allow indexes of any type.

An access specifier of an array property must list a method identifier. In other words, the **read** and **write** specifiers of an array property are not allowed to specify a field name.

The methods listed in array property access specifiers are governed by the following rules:

- The method listed in the **read** specifier of an array property must be a function that takes the same number and types of parameters as are listed in the property's index parameter list, and the function result type must be identical to the property type.
- The method listed in the **write** specifier of an array property must be a procedure with the same number and types of parameters as are listed in the property's index parameter list, plus an additional value or constant parameter of the same type as the property type.

Accessing an array property

You access an array property by following the property identifier with a list of actual parameters enclosed in square brackets. When using array properties you cannot access the array as a whole.

Multiple-index properties

An array property, like an array, can have more than one index. The corresponding parameters to the **read** and **write** methods must still have the same signatures as the indexes, and must appear in the same order as the indexes.

See also

Default array properties
Fields for reading and writing properties
Index specifiers
Property syntax
Read-only and write-only properties

Example

Declaring an array property example

Accessing an array property

Declaring an array property example

The following example declares a property that looks like an array of strings. It also declares the **read** and **write** routines for the property.

```
property MyStrings[Index: Integer]: string read GetMyString write SetMyString;
function GetMyString(Index: Integer): string;
procedure SetMyString(Index: Integer; const NewElement: string);
```

Accessing an array property

Given the following property declaration,

```
property MyStrings[Index: Integer]: string read GetMyString write SetMyString;
```

you can set or retrieve strings as follows:

```
var
  YourString: string;
begin
  YourString := MyStrings[1];
  MyStrings[2] := 'This is a string.';
end;
```

Default array properties

See also

You can declare an array property as the default property, which means you can reference the array without using its name, treating the object itself as if it were indexed. An object can have only one default property.

To declare a default array property, add the directive **default** after an array property:

```
type
  TMyObject = class(TObject)
    property X[Index: Integer]: Integer read GetElement; default;
  end;
```

To access the default array property, place the index next to the object identifier without specifying the property name. For example, given the above declaration, the following code accesses the default property X directly and again as the default property:

```
var
  MyObject: TMyObject;
begin
  MyObject.X[1] := 42;
  MyObject[2] := 1993;
end;
```

Essentially, when an object reference is followed by a list of indexes enclosed in square brackets, the compiler automatically selects the object type's default array property, or issues an error if the object type has no default array property.

If an object defines a default array property, derived object automatically inherit the default array property. It is not possible for a derived object to redeclare or hide the default array property.

See also

Array properties

Property syntax

Index specifiers

See also Example

The definition of a property may optionally include an index specifier. Index specifiers allow a number of properties to share the same access methods. An index specifier consists of the directive index followed by an integer constant with a value between 32767 and 32767.

An access specifier of a property with an **index** specifier must list a method identifier. In other words, the **read** and **write** specifiers of a property with an **index** specifier are not allowed to list a field name.

When accessing a property with an **index** specifier, the integer value specified in the property definition is automatically passed to the access method as an extra parameter. For that reason, an access method for a property with an **index** specifier must take an extra value parameter of type Integer.

For a property **read** function, the extra parameter must be the last parameter.

For a property **write** procedure, the extra parameter must be the second-to-last parameter, that is it must immediately precede the the parameter that specifies the new property value.

See also

Array properties

Property syntax

Example

The following example defines the object TRectangle and declares the index properties Left, Top, Right, and Bottom.

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
    property Left: Longint index 0
      read GetCoordinate write SetCoordinate;
    property Top: Longint index 1
      read GetCoordinate write SetCoordinate;
    property Right: Longint index 2
      read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3
      read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint
      read GetCoordinate write SetCoordinate;
    :
  end;
```

Assuming that Rectangle is an object reference of the TRectangle type defined above, the statement

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage specifiers

See also

The optional stored, default, and nodefault specifiers of a property definition are called storage specifiers. They control certain aspects of the run-time type information that gets generated for **published** properties. Storage specifiers are supported only for normal (non-array) properties.

Storage specifiers have no semantic effects on a property, that is, they do not affect how a property is used in program code. However, the Delphi Visual Class Library uses the information generated by storage specifiers to control filing of a component's properties, that is, the automatic saving and loading of a component's property values in a form file. The **stored** directive controls whether a property is filed, and the **default** and **nodefault** properties control the value that is considered a property's default value.

When saving a component's state, the Delphi Visual Class Library iterates over all of the component's **published** properties. For each property, the result of evaluating the Boolean constant, field, or function method of the **stored** specifier controls whether the property is saved. If the result is False, the property is not saved. If the result is True, the property's current value is compared to the value given in the **default** specifier (if present). If the current value is equal to the default value, the property is not saved. Otherwise, if current value is different from the default value, or if the property has no default value, the property is saved.

See also

Property syntax

Property overrides

See also Example

A property definition that does not include a property interface is called a property override. A property override allows a derived class to change the visibility, access specifiers, and storage specifiers of an inherited property.

In its simplest form, a property override specifies only the reserved word **property** followed by an inherited propery identifier. This form is used to change the visibility of a property. If, for example, a base class defines a property in a **protected** part, a derived class can raise the visibility of the property by declaring a property override in a **public** or **published** part.

A property override can include a **read**, **write**, **stored**, and **default** or **nodefault** specifier. Any such specifier overrides the corresponding inherited specifier. Note that a property override can change an inherited access specifier or add a missing access specifier, but it cannot remove an access specifier.

See also

Access methods
Property syntax
Storage specifiers

Example

The following example illustrate the use of property overrides to change the visibility, access specifiers, and storage specifiers of inherited properties.

```
type
  TBase = class
    :
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    :
  end;

type
  TDerived = class(TBase)
    :
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    :
  end;
```

Read-only and write-only properties

See also

You can make a property read-only or write-only by omitting the **read** or **write** portion of the property declaration. For example, to declare a read-only property, you supply only a **read** method:

```
type
  TAnObject = class(TObject)
    property AProperty: TypeX  read GetAnObject;
  end;
```

By not providing a method for setting the property, you ensure that to the end user, the property is read-only. Any attempt to write to a read-only property or read a write-only property causes a compiler error.

See also

Array properties

Fields for reading and writing properties

Property syntax

Changes in method dispatching

See also

Delphi makes several changes in the way objects dispatch method calls. Previous versions of the compiler allowed static, virtual, and dynamic virtual methods. In the new object model, you can use static, virtual, dynamic, and message-handling messages.

In addition to changes in the kinds of methods you can declare, there are also changes in the implementation of abstract methods and changes in the way you override virtual and dynamic methods.

- Dynamic methods
- Message-handling methods
- Abstract methods
- Override directive
- Virtual constructors

See also

Changes in object declarations

Changes in object use

Properties

Dynamic methods

Dynamic method declarations use the directive dynamic. The compiler assigns its own index (which happens to be a negative number, but it is not a number you will ever need to use).

Dynamic methods work just like virtual methods, but instead of an entry in the class' virtual method table, the dynamic method is dispatched by using its index number. The only noticeable difference between dynamic methods and virtual methods is that dynamic-method dispatch takes somewhat longer.

Message-handling methods

In addition to dynamic methods, Delphi also has a new, specialized form of dynamic method called a message-handling method.

There are three important aspects to message-handling methods:

- Declaring message handlers
- Dispatching messages
- Calling inherited message handlers

Declaring message handlers

Message-handling methods have four distinguishing characteristics:

- They are always procedures.
- They are declared with the **message** directive.
- They take an integer constant as a dynamic index, following **message**.
- They take a single parameter, which must be a **var** parameter.

Message handlers cannot have cdecl, virtual, dynamic, override or abstract methods.

A typical message-handling method declaration looks like this:

```
type
  TMyControl = class(TWinControl)
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
  end;
```

The name of the method and the name and type of the parameter are not important. For example, a descendant type could declare an entirely different handler for the same message:

```
type
  TMyOtherControl = class(TMyControl)
    procedure PaintIt(var Info); message WM_PAINT;
  end;
```

Dispatching messages

The actual message dispatching is done by a method inherited from TObject called Dispatch.

Calling inherited message handlers

Within a message-handler method, you can call the message-handler inherited from the object's ancestor type. Because the name and parameter of the method might vary, you can call the inherited method just by using inherited; you do not have to include the ancestor method's name.

For example, the WMPaint method described above might be implemented as follows:

```
procedure TMyControl.WMPaint(var Message: TWMPaint);
begin
  with Message do
  begin
    ...
    inherited;
    ...
  end;
end;
```

The inherited method called will be the one with the same message index (in this case, WM_PAINT). For example, if the method declared above, TMyOtherControl.PaintIt, included an inherited statement, the method called would be TMyControl.WMPaint.

Default message handler

It is always safe to call **inherited** within a message-handler method. If the ancestor type does not declare a specific message handler for a particular message index, **inherited** calls the TObject method DefaultHandler.

Abstract methods

Example

An abstract method is a virtual or dynamic method whose implementation is not defined in the class declaration in which it appears; its definition is instead deferred to descendant classes. An abstract method defines an interface but not the underlying operation.

A method is abstract if an abstract directive is included in its declaration. A method can be declared **abstract** only if it is first declared **virtual** or **dynamic**.

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.

Calling an abstract method through an object that has not overridden the method will generate an exception at run time.

Example

The following example declares an abstract method.

```
type
  TMyObject = class
    procedure Something; virtual; abstract;
  end;
```

Override directive

Because virtual methods now 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**.

For example,

```
type
  TAnObject = class
    procedure P; virtual;
  end;
  TAnotherObject = class(TAnObject)
    procedure P; override;
  end;
```

Type TAnotherObject could declare its P method with either virtual or dynamic, but those mean something different: either of those would introduce a different method P, replacing, rather than overriding, the inherited P.

Virtual constructors

New-style objects can have virtual constructors. TObject.Create, for instance, is not virtual, but many VCL objects have virtual constructors.

Run-time errors

Certain errors at run time cause the program to display an error message and terminate:

```
Run-time error nnn at xxxxxxxx
```

where nnn is the run-time error number, and xxxxxxxx is the run-time error address.

Delphi applications that use the SysUtils unit map most run-time errors into exceptions, which enable your application to resolve the error without terminating. This is called exception handling.

The run-time errors are divided into three categories:

- I/O errors, numbered 100 through 149
- fatal errors, numbered 200 through 255
- Operating system errors

I/O errors

These errors cause termination if the particular statement was compiled in the {$I+} state. In the {$I-} state, the program continues to execute, and the error is reported by the IOResult function.

Number	Name	Description
100	Disk read error	Reported by Read on a typed file if you attempt to read past the end of the file.
101	Disk write error	Reported by CloseFile, Write, Writeln, or Flush if the disk becomes full.
102	File not assigned	Reported by Reset, Rewrite, Append, Rename, and Erase if the file variable has not been assigned a name through a call to Assign or AssignFile.
103	File not open	Reported by CloseFile, Read, Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead, or BlockWrite if the file is not open.
104	File not open for input	Reported by Read, Readln, Eof, Eoln, SeekEof, or SeekEoln on a text file if the file is not open for input.
105	File not open for output	Reported by Write and Writeln on a text file if you do not generate a Console application.
106	Invalid numeric format	Reported by Read or Readln if a numeric value read from a text file does not conform to the proper numeric format.

Fatal errors

These errors always immediately terminate the program.

In applications that use the SysUtils unit (as most Delphi applications do), these errors are mapped to exceptions. For detailed descriptions of the error conditions that produce each error, see the explanations of the exceptions.

Number	Name	Exception
200	Division by zero	EDivByZero
201	Range check error	ERangeError
202	Stack overflow	EStackOverflow
203	Heap overflow error	EOutOfMemory
204	Invalid pointer operation	EInvalidPointer
205	Floating point overflow	EOverflow
206	Floating point underflow	EUnderflow
207	Invalid floating point operation	EInvalidOp
215	Arithmetic overflow error	EIntOverflow
216	Access violation	EAccessViolation
217	Control-C	EControlC
218	Privileged instruction	EPrivilege
219	Invalid typecast	EInvalidCast
220	Invalid variant typecast	EVariantError
221	Invalid variant operation	EVariantError
222	No variant method call dispatcher	EVariantError
223	Cannot create variant array	EVariantError
224	Variant does not contain array	EVariantError
225	Variant array bounds error	EVariantError
226	TLS initialization error	

Operating system errors

All errors other than I/O errors and fatal errors are reported with the error codes returned by the Win32 error function, GetLastError. The error code values are dependent on the operating system, but you can see a list of them in the Win32 documentation.

Topic Not Found

The topic you are looking for was not found. The problem might be with the Help file or it might be with the link to the topic. You may be able to find this topic by using the Contents, Index or Search tab.

To search for a topic,

1 Click the Contents tab to browse through topics by category.

2 Click the Index tab to see a list of index entries.

Either type the word you're looking for or scroll through the list.

3 Click the Find tab to search for words or phrases

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

This is associated with the Delphi 95 project file document.