Lexical elements

See also

These topics provide a formal definition of the C++Builder lexical elements. They describe the different categories of word-like units (tokens) recognized by a language.

See the topics listed under **See Also** to learn about lexical elements.

The tokens in C++Builder are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

A C++Builder program starts as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the C++Builder editor). The basic program unit in C++Builder is the file. This usually corresponds to a named file located in RAM or on disk and having the extension .C or .CPP.

The preprocessor first scans the program text for special preprocessor *directives* (see Preprocessor directives for details). For example, the directive **#include** <*inc_file*> adds (or *includes*) the contents of the file *inc_file* to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

In the tokenizing phase of compilation, the source code file is *parsed* (that is, broken down) into tokens and whitespace.

Whitespace

See also

Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i;
   float f;
```

are lexically equivalent and parse identically to give the six tokens:

- **int**
- *i*
- ;
- **float**
- **f**
- ;

The ASCII characters representing whitespace can occur within *literal strings*, in which case they are protected from the normal parsing process (they remain as part of the string). For example,

```
char name[] = "Borland International";
```

parses to seven tokens, including the single literal-string token "Borland International"

**Line splicing with **

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \
International"
```

is parsed as "Borland International" (see String constants for more information).

Comments

See also

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. Both are supported by C++Builder, with an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

You should also follow the guidelines on the use of whitespace and delimiters in comments discussed later in this topic to avoid other portability problems.

C comments

A C comment is any sequence of characters placed after the symbol pair /*. The comment terminates at the first occurrence of the pair ***/** following the initial /*. The entire sequence, including the four comment-delimiter symbols, is replaced by one space *after* macro expansion. Note that some C implementations remove comments without space replacements.

C++Builder does not support the nonportable *token pasting* strategy using /**/. Token pasting in C++Builder is performed with the ANSI-specified pair ##, as follows:

```
#define VAR(i,j)  (i/**/j)    /* won't work */
#define VAR(i,j)  (i##j)      /* OK in C++Builder */
#define VAR(i,j)  (i ## j)    /* Also OK */
```

In C++Builder,

```
int /* declaration */ i /* counter */;
```

parses as these three tokens:

```
int   i;
```

See Token Pasting with ## for a description of *token pasting.*

C++ comments

C++ allows a single-line comment using two adjacent slashes (//). The comment can start in any position, and extends until the next new line:

```
class X {  // this is a comment
... };
```

You can also use // to create comments in C code. This is specific to C++Builder.

Nested comments

ANSI C doesn't allow nested comments. The attempt to comment out a line

```
/*  int /* declaration */ i /* counter */; */
```

fails, because the scope of the first **/*** ends at the first */. This gives

```
i ; */
```

which would generate a syntax error.

By default, C++Builder won't allow nested comments, but you can override this with compiler options.

Delimiters and whitespace

In rare cases, some whitespace before /* and //, and after */, although not syntactically mandatory, can avoid portability problems. For example, this C++ code:

```
int i = j//* divide by k*/k;
+m;
```

parses as int i = j +m; not as

```
int i = j/k;
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

Tokens

See also

Tokens are word-like units recognized by a language. C++Builder recognizes six classes of tokens.

Here is the formal definition of a token:

- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *operator*
- *punctuator* (also known as separators)

As the source code is scanned, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, *external* would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier *al.*

See Token Pasting with ## for a description of *token pasting.*

Keywords

See also

Keywords are words reserved for special purposes and must not be used as normal identifier names. See the:

- Alphabetical list of keywords.
- Table of C++ Keywords
- Table of C++Builder Register Pseudovariables

If you use non-ANSI keywords in a program and you want the program to be ANSI compliant, always use the non-ANSI keyword versions that are prefixed with double underscores. Some keywords have a version prefixed with only one underscore; these keywords are provided to facilitate porting code developed with other compilers. For ANSI-specified keywords there is only one version.

Note: Note that the keywords **_ _try** and **try** are an exception to the discussion above. The keyword **try** is required to match the **catch** keyword in the C++ exception-handling mechanism. **try** cannot be substituted by **_ _try.** The keyword **_ _try** can only be used to match the **_ _except** or **_ _finally** keywords. See the discussions on C++ exception handling and C-based structured exceptions for more information.

Table of C++ Specific Keywords

There are several keywords specific to C++. They are not available if you are writing a C-only program.

The keywords specific to C++ are:

asm	mutable	this
bool	namespace	throw
catch	new	true
class	operator	try
const_cast	private	typeid
delete	explicit	protected reinterpret_cast
dynamic_cast	public	using
false	__rtti	virtual
friend	static_cast	wchar_t
inline	template	typename

Table of C++Builder register pseudovariables

See also

_AH	**_CL**	**_EAX**1	**_ESP**
_AL	**_CS**	**_EBP**1	**_FLAGS**
_AX	**_CX**	**_EBX**1	**_FS**
_BH	**_DH**	**_ECX**1	**_GS**1
_BL	**_DI**	**_EDI**1	**_SI**
_BP	**_DL**	**_EDX**1	**_SP**
_BX	**_DS**	**_ES**	**_SS**
_CH	**_DX**	**_ESI**1	

1 These pseudovariables are always available to the 32-bit compiler. The 16-bit compiler can use these only whe you use the option to generate 80386 instructions.

All but the _FLAGS register pseudovariable are associated with the general purpose, segment, address, and special purpose registers.

Use register pseudovariables anywhere that you can use an integer variable to directly access the corresponding 80x86 register.

The 16-bit flags register contains information about the state of the 80x86 and the results of recent instructions.

C++Builder Keyword Extensions

C++Builder provides additional keywords that are not part of the ANSI Standard.

The C++Builder keyword extensions are:

_asm

__asm

__automated

_cdecl

cdecl

__classid

__closure

__declspec

__except

__export

_export

__fastcall

_fastcall

__finally

_import

__import

__int8

__int16

__int32

__int64

_pascal

__pascal

pascal

__property

__published

__rtti

__thread

__try

Identifiers

See also

Here is the formal definition of an identifier:

identifier:

nondigit

identifier nondigit

identifier digit

nondigit: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z _

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. (Identifiers can contain the letters *a* to *z* and *A* to *Z*, the underscore character "_", and the digits 0 to 9.) There are only two restrictions:

- The first character must be a letter or an underscore.
- By default, C++Builder recognizes only the first 32 characters as significant. The number of significant characters can be command-line options, but not increased.

Case sensitivity

C++Builder identifiers are case sensitive, so that *Sum*, *sum* and *suM* are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, C++Builder offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. With the case-insensitive option, the globals *Sum* and *sum* are considered identical, resulting in a possible. "`Duplicate symbol`" warning during linking.

An exception to these rules is that identifiers of type **_ _pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same *scope* and sharing the same *name space*. Duplicate names are legal for *different* name spaces regardless of scope rules.

Constants

See also

Constants are tokens representing fixed numeric or character values.

C++Builder supports four classes of constants: integer, floating point, character (including strings), and enumeration.

Internal representation of numerical types shows how these types are represented internally.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in the following table.

Constants: Formal Definitions

constant:
- *floating-constant*
- *integer-constant*
- *numeration-constant*
- *character-constant*

floating-constant:
- *fractional-constant <exponent-part> <floating-suffix>*
- *digit-sequence exponent-part <floating-suffix>*

fractional-constant:
- *<digit-sequence>* **.** *digit-sequence*
- *digit-sequence* **.**

exponent-part:
- e *<sign> digit-sequence*
- E *<sign> digit-sequence*

sign: one of
- + -

digit-sequence:
- *digit*
- digit-sequence digit

floating-suffix: one of
- f l F L

integer-constant:
- *decimal-constant <integer-suffix>*
- *octal-constant <integer-suffix>*
- *hexadecimal-constant <integer-suffix>*

decimal-constant:
- *nonzero-digit*
- *decimal-constant digit*

octal-constant:
- 0

nonzero-digit: one of
- 1 2 3 4 5 6 7 8 9

octal-digit: one of
- 0 1 2 3 4 5 6 7

hexadecimal-digit: one of
- 0 1 2 3 4 5 6 7 8 9
- a b c d e f
- A B C D E F

integer-suffix:
- *unsigned-suffix <long-suffix>*
- *long-suffix <unsigned-suffix>*

unsigned-suffix: one of
- u U

long-suffix: one of
- l L

enumeration-constant:
- *identifier*

character-constant
- *c-char-sequence*

c-char-sequence:
- *c-char*
- *c-char-sequence c-char*

c-char:
- Any character in the source character set

octal-constant *octal-digit*

hexadecimal-constant:

0 x hexadecimal-digit

0 X *hexadecimal-digit*

hexadecimal-constant *hexadecimal-digit*

except the single-quote ('), backslash (\\), or

newline character *escape-sequence.*

escape-sequence: one of the following

	\\"	\\'	\\?	\\\\
	\\a	\\b	\\f	\\n
\\o	\\oo	\\ooo	\\r	
\\t	\\v	\\Xh...	\\xh...	

Integer constants

See also

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in C++Builder integer constants without L or U.. Note that the rules vary between decimal and nondecimal constants.

Decimal

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10;  /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0;   /*decimal 0 = octal 0 */
```

Octal

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 are truncated.

Hexadecimal

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF are truncated.

long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces the constant to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both *L* and *U* suffixes on the same constant in any order or case: *ul, lu, UL*, and so on. See the table of Borland constants.

The data type of a constant in the absence of any suffix (*U, u, L,* or *l*) is the first of the following types that can accommodate its value:

Decimal	**int, long int, unsigned long int**
Octal	**int, unsigned int, long int, unsigned long int**
Hexadecimal	**int, unsigned int, long int, unsigned long int**

If the constant has a *U* or *u* suffix, its data type will be the first of **unsigned int, unsigned long int** that can accommodate its value.

If the constant has an *L* or *l* suffix, its data type will be the first of **long int, unsigned long int** that can accommodate its value.

If the constant has both u and l suffixes, (*ul, lu, Ul, lU, uL, Lu, LU or UL*), its data type will be **unsigned long int**.

C++Builder integer constants without L or U summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Extended integer types

See also

You can specify the size for integer types. You must use the appropriate suffix when using extended integers.

Type	Suffix	Example	Storage
__int8	i8	__int8 c = 127i8;	8 bits
__int16	i16	__int16 s = 32767i16;	16 bits
__int32	i32	__int32 i = 123456789i32;	32 bits
__int64	i64	__int64 big = 12345654321i64;	64 bits
unsigned __int64	ui64	unsigned __int64 hugeInt = 1234567887654321ui64;	64 bits

C++Builder integer constants without L or U

See also

Decimal constants

0	to	32,767	**int**
32,768	to	2,147,483,647	**long**
2,147,483,648	to	4,294,967,295	**unsigned long**
		> 4294967295	truncated

Octal constants

00	to	077777	**int**
010000	to	0177777	**unsigned int**
02000000	to	017777777777	**long**
020000000000	to	037777777777	**unsigned long**
		> 037777777777	truncated

Hexadecimal constants

0x0000	to	0x7FFF	**int**
0x8000	to	0xFFFF	**unsigned int**
0x10000	to	0x7FFFFFFF	**long**
0x80000000	to	0xFFFFFFFF	**unsigned long**
		>0xFFFFFFFF	truncated

Floating-point constants

See also

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- *e* or *E* and a signed integer exponent (optional)
- Type suffix: *f* or *F* or *l* or *L* (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter *e* (or *E*) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Here are some examples:

Constant	Value
23.45e6	23.45 106
.0	0
0.	0
1.	1.0 100 = 1.0
-1.23	-1.23
2e-5	2.0 10-5
3E+10	3.0 1010
.09E34	0.09 1034

In the absence of any suffixes, floating-point constants are of type **double.** However, you can coerce a floating constant to be of type **float** by adding an *f* or *F* suffix to the constant. Similarly, the suffix *l* or *L* forces the constant to be data type **long double**. The table below shows the ranges available for **float**, **double**, and **long double**.

C++Builder floating-point constant sizes and ranges

Type	Size (bits)	Range
float	32	3.4 10_{-38} to 3.4 10_{38}
double	64	1.7 10_{-308} to 1.7 10_{308}
long double	80	3.4 10_{-4932} to 1.1 10_{4932}

Character constants

See also

A *character constant* is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-charactrer constants have data type **int**. The number of bits used to internally represent a character constant is **sizeof**(**int**). In a 16-bit program, the upper byte is zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

To learn more about character constants, see the following topics.

- Three char types
- Escape sequences
- Wide-character and multi-character constants

Note: To compare sizes of character types, compile this as a C program and then as a C++ program.

```
#include <stdio.h>
#define CH 'x'       /* A CHARACTER CONSTANT */
void main(void) {
   char ch = 'x';    /* A char VARIABLE      */
   printf("\nSizeof int     = %d", sizeof(int) );
   printf("\nSizeof char    = %d", sizeof(char) );
   printf("\nSizeof ch      = %d", sizeof(ch) );
   printf("\nSizeof CH      = %d", sizeof(CH) );
   printf("\nSizeof wchar_t = %d", sizeof(wchar_t) );
}
```

Note: Sizes are in bytes.

Sizes of character types

Output when compiled as C program			**Output when compiled as C++ program**		
Sizeof int	=	4	Sizeof int	=	4
Sizeof char	=	1	Sizeof char	=	1
Sizeof ch	=	1	Sizeof ch	=	1
Sizeof CH	=	4	Sizeof CH	=	1
Sizeof wchar_t	=	2	Sizeof wchar_t	=	2

The three char types

See also

One-character constants, such as 'A', '\t' and '007', are represented as **int** values. In this case, the low-order byte is *sign extended* into the high bit; that is, if the value is greater than 127 (base 10), the upper bit is set to -1 (=0xFF). This can be disabled by declaring that the default **char** type is **unsigned**, which forces the high bit to be zero regardless of the value of the low bit.

The three character types, **char**, **signed char**, and **unsigned char,** require an 8-bit (one byte) storage. In C and C++Builder programs prior to version Borland C++ 4.0 , **char** is treated the same as **signed char**. The behavior of C programs is unaffected by the distinction between the three character types.

Note: To retain the old behavior, use the -K2 command-line option and Borland C++ 3.1 header files and libraries.

In a C++ program, a function can be overloaded with arguments of type **char**, **signed char**, or **unsigned char**. For example, the following function prototypes are valid and distinct:

```
void func(char ch);
void func(signed char ch);
void func(unsigned char ch);
```

If only one of the above prototypes exists, it will accept any of the three character types. For example, the following is acceptable:

```
void func(unsigned char ch);
void main(void) {
  signed char ch = 'x';
  func(ch);
  }
```

Escape sequences

See also

The backslash character (\) is used to introduce an *escape sequence*, which allows the visual representation of certain nongraphic characters. For example, the constant **\n** is used to the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for *Ctrl-C* or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff for C++Builder). Larger numbers generate the compiler error Numeric constant too large. For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Take this example.

```
printf("\x0072.1A Simple Operating System");
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, C++Builder compiles it as the hexadecimal number \x0072 and the literal string "2.1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

The following table shows the available escape sequences.

C++Builder escape sequences

Note: You must use \\ to represent an ASCII backslash, as used in operating system paths.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O=a string of up to three octal digits
\xH		any	H=a string of hex digits
\XH		any	H=a string of hex digits

Wide-character and multi-character constants

See also

Wide-character types can be used to represent a character that does not fit into the storage space allocated for a **char** type. A wide character is stored in a two-byte space. A character constant preceded immediately by an *L* is a wide-character constant of data type *wchar_t* (defined in stddef.h). For example:

```
wchar_t ch = L'AB';
```

When *wchar_t* is used in a C program it is a type defined in stddef.h header file. In a C++ program, **wchar_t** is a keyword that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In C++, **wchar_t** is the same size, signedness, and alignment requirement as an **int** type.

A string preceded immediately by an *L* is a wide-character string. The memory allocation for a string is two bytes per character. For example:

```
wchar_t str = L"ABCD";
```

Multi-character constants

C++Builder also supports multi-character constants. Multi-character constants can consist of as many as four characters. For example, the constant, '\006\007\008\009' is valid only in an C++Builder program. Multi-character constants are always 32-bit **int** values. The constants are not portable to other C compilers.

String constants

See also

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of-**char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written "".

The characters inside the double quotes can include escape sequences. This code, for example:

```
"\t\t\"Name\"\\\tAddress\n\n"
```

prints like this:

```
"Name"\       Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The \" provides interior double quotes.

If you compile with the -**A** option for ANSI compatibility, the escape character sequence "\\", is translated to "\" by the compiler.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>
int main() {
   char    *p;

   p = "This is an example of how C++Builder"
     " will\nconcatenate very long strings for you"
     " automatically, \nresulting in nicer"
      " looking programs.";
   printf(p);
   return(0);
}
```

The output of the program is

```
This is an example of how C++Builder will
concatenate very long strings for you automatically,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

Enumeration constants

See also

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed. See Enumerations and enum (keyword) for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional *initializers*. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, *cubs,* and *dodgers* are enumeration constants of type *team* that can be assigned to any variables of type *team* or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

Constants and internal representation

See also

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For C++Builder, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of internal representations for the various data types.

The following tables list the sizes and resulting ranges of the data types for C++Builder. Internal representation of numerical types shows how these types are represented internally.

32-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned int	32	0 to 4,294,967,295	Large numbers and loops
int	32	-2,147,483,648 to 2,147,483,647	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
enum	32	-2,147,483,648 to 2,147,483,647	Ordered sets of values
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4 ^ 10-38 to 1.7 ^ 1038	Scientific (7-digit) precision)
double	64	1.7 ^ 10-308 to 3.4 ^ 10308	Scientific (15-digit precision)
long double	80	3.4 ^ 10-4932 to 1.1 ^ 104932	Financial (18-digit precision)

Data Types (32-bit)

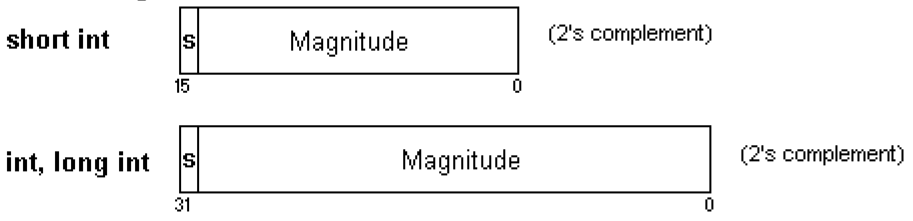
See also Keywords

Type	Length	Range
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
short int	16 bits	-32,768 to 32,767
unsigned int	32 bits	0 to 4,294,967,295
int	32 bits	-2,147,483,648 to 2,147,483,647
unsigned long	32 bits	0 to 4,294,967,295
enum	16 bits	-2,147,483,648 to 2,147,483,647
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4×10^{-38} to $3.4 \times 10^{+38}$
double	64 bits	1.7×10^{-308} to $1.7 \times 10^{+308}$
long double	80 bits	3.4×10^{-4932} to $1.1 \times 10^{+4932}$

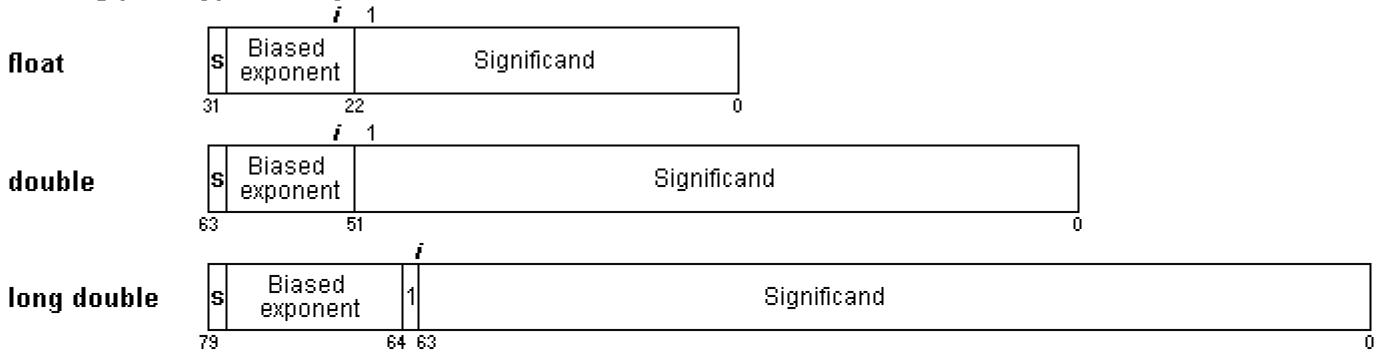
Internal representation of numerical types

See also

32-bit integers



Floating-point types, always



s	=	Sign bit (0 = positive, 1 = negative)	Exponent bias (normalized values):	
i	=	Position of implicit binary point	**float:**	127 (7FH)
1	=	Integer bit of significance:	**double:**	1,023 (3FFH)
		Stored in **long double** Implicit in **float**, **double**	**long double:**	16,383 (3FFFH)

Constant expressions

See also

A constant expression is an expression that always evaluates to a constant (and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is:

```
constant-expression:
  Conditional-expression
```

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- Assignment
- Comma
- Decrement
- Function call
- Increment

Punctuators

See also

The C++Builder punctuators (also known as separators) are:

[ ]

( )

{ }

,

;

:

...

*

=

#

Most of these punctuators also function as operators.

Brackets

Open and close brackets indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];           /* 3 x 4 matrix */
ch = str[3];             /* 4th element */
   .
   .
   .
```

Parentheses

Open and close parentheses **( )** are used to group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);  /* override normal precedence */
if (d == z) ++x;     /* essential with conditional statement */
func();  /* function call, no args */
int (*fptr)();       /* function pointer declaration */
fptr = func;         /* no () means func pointer */
void func2(int n);   /* function declaration with parameters */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered in Expressions.

Braces

Open and close braces **{ }** indicate the start and end of a compound statement:

```
if (d == z)
{
   ++x;
   func();
}
```

The closing brace serves as a terminator for the compound statement, so a ; (semicolon) is not required after the }, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
   {};                  /*illegal semicolon*/
```

```
else
```

Comma

The comma (**,**) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in *comma expressions*. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them:

```
func(i, j);                              /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5));  /* also calls func with two args! *
  /
```

Semicolon

The semicolon (**;**) is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an *expression statement*. The expression is evaluated and its value is discarded. If the expression statement has no side effects, C++Builder might ignore it.

```
a + b;    /* maybe evaluate a + b, but discard value */
++a;      /* side effect on a, but discard value of ++a */
;         /* empty expression = null statement */
```

Semicolons are often used to create an *empty statement:*

```
for (i = 0; i < n; i++)
{
   ;
}
```

Colon

Use the colon (**:**) to indicate a labeled statement:

```
start:    x=0;
   ƒ
goto start;
```

Labels are discussed in Labeled statements.

Ellipsis

The ellipsis (**...**) is three successive periods with no intervening whitespace. Ellipses are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that *func* will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.

In C++, you can omit the comma before the ellipsis.

Asterisk (pointer declaration)

The asterisk (*****) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr;  /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;         /* a pointer to an integer array */
double ***double_ptr;  /* a pointer to a matrix of doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
```

```
a = b * 3.14;
```

Equal sign (initializer)

The equal sign (**=**) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };
int  x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... }  /* Parameter i has default value of zero */
```

The equal sign is also used as the assignment operator in expressions:

```
int a, b, c;
a = b + c;
float *ptr = (float *) malloc(sizeof(float) * 100);
```

Pound sign (preprocessor directive)

The pound sign (**#**) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See Preprocessor directives for more on the preprocessor directives.

and **##** (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase.

Language structure

See also

These topics provide a formal definition of C++ language and its implementation in C++Builder. They describe the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

Declarations

See also

This section briefly reviews concepts related to declarations: objects, storage classes, types, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Objects

See also

An *object* is an identifiable region of memory that can hold a fixed or variable value (or set of values). (This use of the word *object* is different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a *data type*). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely references the object. The type is used

- to determine the correct memory allocation required initially.
- to interpret the bit patterns found in the object during subsequent accesses.
- in many type-checking situations, to ensure that illegal assignments are trapped.

C++Builder supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in memory.

The C++Builder standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that C++Builder can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Objects and declarations

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as *defining declarations*, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Other declarations, known as *referencing declarations*, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its *declaration point* in the source code. Legal exceptions to this rule (known as *forward references*) are labels, calls to undeclared functions, and class, struct, or union tags.

lvalues

An *lvalue* is an object locator: an expression that designates an object. An example of an lvalue expression is **P*, where *P* is any expression evaluating to a non-null pointer. A *modifiable lvalue* is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is *not* a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the *l* stood for "left," meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if *a* and *b* are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as *a = 1*; and *b = a + b* are legal.

rvalues

The expression *a + b* is not an lvalue: *a + b = a* is illegal because the expression on the left is not related to an object. Such expressions are often called *rvalues* (short for right values).

Storage classes and types

See also

Associating identifiers with objects requires each identifier to have at least two attributes: *storage class* and *type* (sometimes referred to as data type). The C++Builder compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location (data segment, register, heap, or stack) of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type. See sizeof for more on this operator.

Scope

See also

The scope of an identifier is that part of the program in which the identifier can be used to access its object. There are five categories of scope: *block* (or *local*), *function*, *function prototype*, *file*, and *class* (C++ only). These depend on how and where identifiers are declared.

- **Block**. The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the *enclosing* block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.
- **Function**. The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing *label_name:* followed by a statement. Label names must be unique within a function.
- **Function prototype**. Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.
- **File**. File scope identifiers, also known as *globals*, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.
- **Class** (C++). A class is a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class.Classes and their objects have many special access and scoping rules; see Classes.
- **Condition** (C++). Declarations in conditions are supported. Variables can be declared within the expression of **if**, **while**, and **switch** statements. The scope of the variable is that of the statement. In the case of an **if** statement, the variable is also in scope for the **else** block.

Name spaces

Name space is the scope within which an identifier must be unique. C uses four distinct classes of identifiers:

- **goto** label names. These must be unique within the function in which they are declared.
- Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all.
- Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
- Variables, **typedef**s, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Note: Structures, classes, and enumerations are in the same name space in C++.

Visibility

See also

The *visibility* of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily *hidden* by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Note: Visibility cannot exceed scope, but scope can exceed visibility.

```
    .
    .
    .
{
   int i; char ch;  // auto by default
   i = 3;           // int i and char ch in scope and visible
    .
    .
    .

   {
      double i;
      i = 3.0e3;    // double i in scope and visible
                    // int i=3 in scope but hidden
      ch = 'A';     // char ch in scope and visible
   }
                    // double i out of scope
   i += 1;          // int i visible and = 4
    .
    .
    .
// char ch still in scope & visible = 'A'
}
    .
    .
    .
// int i and char ch out of scope
```

Again, special rules apply to hidden class names and class member names: C++ operators allow hidden identifiers to be accessed under certain conditions

Duration

See also

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedef**s and types, have real memory allocated during run time. There are three kinds of duration: *static*, *local*, and *dynamic*.

Static

Memory is allocated to objects with *static* duration as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory model in force, although in 32-bit development, only the flat memory model is supported. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, constructor.

Don't confuse static duration with file or global scope. An object can have static duration and local scope

Local

Local duration objects, also known as *automatic* objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects must always have local or function scope. The storage class specifier **auto** can be used when declaring local duration variables, but is usually redundant, because **auto** is the default for variables declared within a block. An object with local duration also has local scope, because it does not *exist* outside of its enclosing block. The converse is not true: a local scope object can have static duration.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. C++Builder can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Note: The C++Builder compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

Dynamic

Dynamic duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the heap, using either standard library functions such as *malloc*, or by using the C++ operator **new**. The corresponding deallocations are made using *free* or **delete**.

static

Example Keywords

Syntax

```
static <data definition> ;
static <function name> <function definition> ;
```

Description

Use the **static** storage class specifier with a local variable to preserve the last value between successive calls to that function. A **static** variable acts like a local variable but has the lifetime of an external variable.

In a class, data and member functions can be declared **static**. Only one copy of the **static** data exists for all objects of the class.

A **static** member function of a global class has external linkage. A member of a local class has no linkage. A **static** member function is associated only with the class in which it is declared. Therefore, such member functions cannot be **virtual**.

Static member functions can only call other **static** member functions and only have access to **static** data. Such member functions do not have a **this** pointer.

Translation units

See also

The term *translation unit* refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

```
translation-unit:
  external-declaration
  translation-unit external-declaration
external-declaration
  function-definition
  declaration
```

word *external* has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the section Linkage.) Any declaration that also reserves storage for an object or function is called a definition (or defining declaration). For more details, see External declarations and definitions.

Linkage

See also

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with *external linkage* represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with *internal linkage* represents the same object or function within one file only. Identifiers with *no linkage* represent unique entities.

External and internal linkage rules

- Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.
- For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.
- If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
- If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
- If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

Identifiers with no linkage attribute:

- Any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
- Function parameters
- Block scope identifiers for objects declared without the storage class specifier **extern**

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or .OBJ files compiled with a C compiler

To tell the C++ compiler not to mangle the name of a function, declare the function as `extern "C"`, like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler that references to the function *Cfunc* should not be mangled.

You can also apply the `extern "C"` declaration to a block of names:

```
extern "C" {
   void Cfunc1( int );
   void Cfunc2( int );
   void Cfunc3( int );
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions *Cfunc1*, *Cfunc2*, and *Cfunc3* should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {
   #include "locallib.h"
};
```

Introduction to declaration syntax

See also

All six interrelated attributes (storage classes, types, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations*.

Declarations can be *defining declarations* (also known as *definitions*) or *referencing declarations* (sometimes known as *nondefining declarations*). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Tentative definitions

See also

The ANSI C standard supports the concept of the *tentative definition*. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;
int x;           /*legal, one copy of x is reserved */
int y;
int y = 4;       /* legal, y is initialized to 4 */
int z = 5;
int z = 6;       /* not legal, both are initialized definitions */
```

Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

Possible declarations

See also

The range of objects that can be declared includes

- Variables
- Functions
- Classes and class members (C++)
- Types
- Structure, union, and enumeration tags
- Structure members
- Union members
- Arrays of other types
- Enumeration constants
- Statement labels
- Preprocessor macros

The full syntax for declarations is shown in Tables 2.1 through 2.3. The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use **typedefs** to improve legibility.

In Borland C++ declaration syntax, note the restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail in Variable Modifiers and Function Modifiers.

C++Builder declaration syntax

declaration:

<decl-specifiers> <declarator-list>;

asm-declaration

function-declaration

linkage-specification

decl-specifier:

storage-class-specifier

type-specifier

function-specifier

friend (C++ specific)

typedef

decl-specifiers:

<decl-specifiers> decl-specifier

storage-class-specifier:

auto

register

static

extern

function-specifier: (C++ specific)

inline

virtual

simple-type-name:

class-name

***typedef**-name*

elaborated-type-specifier:

class-key identifier

class-key class-name

enum *enum-name*

class-key: (C++ specific)

class

struct

union

enum-specifier:

enum *<identifier> { <enum-list> }*

enum-list:

enumerator

enumerator-list , enumerator

enumerator:

identifier

identifier = constant-expression

constant-expression:

conditional-expression

linkage-specification: (C++ specific)

extern *string { <declaration-list> }*

extern *string declaration*

type-specifier:

simple-type-name

class-specifier

boolean

  char

  short

  int

__int8

__int16

__int32

__int64

  long

  signed

  unsigned

  float

  double

  void

declarator-list:

  init-declarator

  declarator-list , init-declarator

init-declarator:

  declarator <initializer>

declarator:

  dname

  modifier-list

  pointer-operator declarator

  declarator (parameter-declaration-list)

    <cv-qualifier-list >

    (The <cv-qualifier-list > is for C++ only.)

  declarator [<constant-expression>]

  (declarator)

modifier-list:

  modifier

  modifier-list modifier

modifier:

  _ _cdecl

  _ _pascal

_ _stdcall

expression

_ _fastcall

pointer-operator:

  ** <cv-qualifier-list>*

  enum-specifier

  elaborated-type-specifier

  const

  volatile

declaration-list:

  declaration

  declaration-list ; declaration

type-name:

  type-specifier <abstract-declarator>

abstract-declarator:

  pointer-operator <abstract-declarator>

  <abstract-declarator> (argument-declaration-list)

  <cv-qualifier-list>

  <abstract-declarator> [<constant-expression>]

  (abstract-declarator)

argument-declaration-list:

  <arg-declaration-list>

  arg-declaration-list , ...

  <arg-declaration-list> **...** (C++ specific)

arg-declaration-list:

  argument-declaration

  arg-declaration-list , argument-declaration

argument-declaration:

  decl-specifiers declarator

  decl-specifiers declarator = expression

    (C++ specific)

  decl-specifiers <abstract-declarator>

    decl-specifiers <abstract-declarator> **=**

    (C++ specific)

  function-definition:

  function-body:

  compound-statement

initializer:

    & <cv-qualifier-list> (C++ specific)

    *class-name :: * <cv-qualifier-list>*

      (C++ specific)

cv-qualifier-list:

    cv-qualifier <cv-qualifier-list>

cv-qualifier

    const

    volatile

dname:

    name

    class-name (C++ specific)

    ~ *class-name* (C++ specific)

    type-defined-name

    = *expression*

    = { initializer-list }

    (*expression-list*) (C++ specific)

initializer-list:

    expression

    initializer-list , expression

    { initializer-list <,> }

External declarations and definitions

See also

The storage class specifiers auto and register cannot appear in an external declaration. For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of sizeof), then exactly one external definition of that identifier must be somewhere in the entire program.

C++Builder allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. Here's an example:

```
int a[];              // no size
struct mystruct;      // tag only, no member declarators
   .
   .
   .
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
   int i, j;
};                    // add member declarators
```

C++Builder class declaration syntax (C++ only) covers class declaration syntax. In the section on classes (beginning with Classes), you can find examples of how to declare a class. Referencing covers C++ reference types (closely related to pointer types) in detail. Finally, see Using Templates for a discussion of **template**-type classes.

C++Builder class declaration syntax (C++ only)

class-specifier:

  class-head { <member-list> }

class-head:

  class-key <identifier> <base-specifier>

  class-key class-name <base-specifier>

member-list:

  member-declaration <member-list>

  access-specifier : <member-list>

member-declaration:

  <decl-specifiers> <member-declarator-list> ;

  function-definition <;>

  qualified-name ;

member-declarator-list:

  member-declarator

  member-declarator-list, member-declarator

member-declarator:

  declarator <pure-specifier>

  <identifier> : constant-expression

pure-specifier:

  = 0

base-specifier:

  : base-list

base-list:

base-specifier

  base-list , base-specifier

base-specifier:

  class-name

  virtual *<access-specifier> class-name*

  access-specifier <**virtual**> *class-name*

access-specifier:

  private

  protected

  public

conversion-function-name:

  operator *conversion-type-name*

conversion-type-name:

  type-specifiers <pointer-operator>

constructor-initializer:

  : *member-initializer-list*

member-initializer-list:

  member-initializer

  member-initializer , member-initializer-list

member-initializer:

  class name (*<argument-list>*)

  identifier (*<argument-list>*)

operator-function-name:

  operator *operator-name*

operator-name: one of

	new	delete	sizeof	typeid	
+	-	*	/	%	^
&	|	~	!	=	<>
+=	-=	=*	/=	%=	^=
&=	|=	<<	>>	>>=	<<=
==	!=	<=	>=	&&	||
++	__	,	->*	->	()
[ ]	.*				

Type Specifiers

See also

The type determines how much memory is allocated to an object and how the program interprets the bit patterns found in the object's storage allocation. A data type is the set of values (often implementation-dependent) identifiers can assume, together with the set of operations allowed on those values.

The *type specifier* with one or more optional *modifiers* is used to specify the type of the declared identifier:

```
int i;                   // declare i as an integer
unsigned char ch1, ch2;  // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++, a missing type specifier can lead to syntactic ambiguity, so C++ practice requires you to explicitly declare all **int** type specifiers.

The type specifier keywords in C++Builder are:

char	float	signed	wchar_t
class	int	struct	
double	long	union	
enum	short	unsigned	

Use the sizeof operators to find the size in bytes of any predefined or user-defined type.

Type categories

See also

The four basic type categories (and their subcategories) are as follows:

- Aggregate
- Array
- **struct**
- **union**
- **class** (C++ only)
- Function
- Scalar
- Arithmetic
- Enumeration
- Pointer
- Reference (C++ only)
- void)

Types can also be viewed in another way: they can be *fundamental* or *derived* types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.

A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes

Given any nonvoid type **type** (with some provisos), you can declare derived types as follows:

Declaring types

Declaration	Description
type *t;*	An object of type ***type***
type *array*[10];	Ten ***type***s: *array*[0] - *array*[9]
type **ptr;*	*ptr* is a pointer to ***type***
type *&ref = t;*	*ref* is a reference to ***type*** (C++)
type *func*(**void**);	*func* returns value of type ***type***
void *func1*(***type*** *t*);	*func1* takes a type ***type*** parameter
struct st {***type*** *t1*; ***type*** *t2*};	structure **st** holds two ***type***s

Note: type& var, type &var, and type & var are all equivalent.

void

Example Keywords

Syntax

```
void identifier
```

Description

void is a special type indicating the absence of any value. Use the **void** keyword as a function return type if the function does not return a value.

```
void hello(char *name)
{
  printf("Hello, %s.",name);
}
```

Use **void** as a function heading if the function does not take any parameters.

```
int init(void)
{
  return 1;
}
```

Void Pointers

Generic pointers can also be declared as **void**, meaning that they can point to any type.

void pointers cannot be dereferenced without explicit casting because the compiler cannot determine the size of the pointer object.

The fundamental types

See also

The fundamental type specifiers are built from the following keywords:

char	**__int8**	**long**
double	**__int16**	**signed**
float	**__int32**	**short**
int	**__int64**	**unsigned**

From these keywords you can build the integral and floating-point types, which are together known as the *arithmetic* types. The modifiers **long**, **short**, **signed**, and **unsigned** can be applied to the integral types. The include file limits.h contains definitions of the value ranges for all the fundamental types.

Integral types

char, **short**, **int**, and **long**, together with their unsigned variants, are all considered *integral* data types. Integral types shows the integral type specifiers, with synonyms listed on the same line.

Integral types

char, signed char	Synonyms if default **char** set to **signed.**
unsigned char	
char, unsigned char	Synonyms if default **char** set to **unsigned.**
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Note: These synonyms are not valid in C++. See The three char types.

signed or **unsigned** can only be used with **char**, **short**, **int**, or **long**. The keywords **signed** and **unsigned**, when used on their own, mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is assumed for integral types. An exception arises with **char**. C++Builder lets you set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration `char ch` declares *ch* as **unsigned**. You would need to use `signed char ch` to override the default. Similarly, with a **signed** default for **char**, you would need an explicit `unsigned char ch` to declare an **unsigned char**.

Only **long** or **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to indicate that **short**, **int**, and **long** form a nondecreasing sequence with **"short <= int <= long."** All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

In a C++Builder 32-bit program, the types **int** and **long** are equivalent, both being 32 bits. The signed varieties are all stored in two's complement format using the most significant bit (MSB) as a sign bit: 0 for positive, 1 for negative (which explains the ranges shown in 32-bit data types, sizes, and ranges). In the unsigned versions, all bits are used to give a range of 0 - (2*n* - 1), where *n* is 8, 16, or 32.

Floating-point types

The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. C++Builder uses the IEEE floating-point formats.See the topic on ANSI implementation-specific.

float and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double** *test_case*, for example.

The table of 32-bit data types, sizes, and ranges indicates the storage allocations for the floating-point types

Standard arithmetic conversions

When you use an arithmetic expression, such as *a* + *b*, where *a* and *b* are different arithmetic types, C++Builder performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps C++Builder uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in Methods used in standard arithmetic conversions. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers), or they are of type **double**, **float**, or **long double**.
2. If either operand is of type **long double**, the other operand is converted to **long double**.
3. Otherwise, if either operand is of type **double**, the other operand is converted to **double**.
4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
5. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Methods used in standard arithmetic conversions

Type	Converts to	Method
char	**int**	Zero or sign-extended (depends on default char type)
unsigned char	**int**	Zero-filled high byte (always)
signed char	**int**	Sign-extended (always)
short	**int**	Same value; sign extended
unsigned short	**unsigned int**	Same value; zero filled
enum	**int**	Same value

Special char, int, and enum conversions

Note: The conversions discussed in this section are specific to C++Builder.

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int.**

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign-extends or zero-fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

Initialization

See also

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

- To zero if it is an arithmetic type
- To null if it is a pointer type

Note: If the object has automatic storage duration, its value is indeterminate.

Syntax for initializers

initializer

= expression

= {initializer-list} <,>}

(expression list)

initializer-list

expression

initializer-list, expression

{initializer-list} <,>}

Rules governing initializers

- The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.
- The item to be initialized must be an object (for example, an array) of unknown size.
- For C (not required for C++), all expressions must be constants if they appear in one of these places:
- In an initializer for an object that has static duration.
- In an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed).
- If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
- If a brace-enclosed list has fewer initializers than members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- An initializer list (as described in Arrays, structures, and unions).
- A single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array *days*, which counts how many times each day of the week appears in a month (assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

The following rules initialize character arrays and wide character arrays:

- You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
   char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for *name*[0]), 'n' (for *name*[1]), and so on (and including a null terminator).

- You can initialize a wide character array (one that is compatible with *wchar_t*) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {
   int i;
   char str[21];
   double d;
   } s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces.

Declarations and declarators

See also

A *declaration* is a list of names. The names are sometimes referred to as *declarators* or *identifiers*. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon.

Simple declarations of variable identifiers have the following pattern:

```
data-type var1 <=init1>, var2 <=init2>, ...;
```

where *var1*, *var2*,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type *data-type*. For example,

```
int x = 1, y = 2;
```

creates two integer variables called *x* and *y* (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.

In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions

The format of the declarator indicates how the declared *name* is to be interpreted when used in an expression. If ***type*** is any type, and *storage class specifier* is any storage class specifier, and if *D1* and *D2* are any two declarators, then the declaration

```
storage-class-specifier type D1, D2;
```

indicates that each occurrence of D1 or D2 in an expression will be treated as an object of type ***type*** and storage class *storage class specifier*. The type of the *name* embedded in the declarator will be some phrase containing ***type***, such as "***type***," "pointer to ***type***," "array of ***type***," "function returning ***type***," or "pointer to function returning ***type***," and so on.

For example, in Declaration syntax examples each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single **int** object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Declaration syntax examples

Declarator syntax	Implied *type of name*	Example
type `name;`	*type*	**int** `count;`
type `name[];`	(`open`) array of ***type***	**int** `count[];`
type `name[3];`	Fixed array of three elements,	**int** `count[3];` all of **type** (*name*[0], *name*[1], and *name*[2]
type `*name;`	Pointer to **type**	**int** `*count;`
type `*name[];`	(`open`) array of pointers to **type**	**int** `*count[];`
type `*(name[]);`	Same as above	**int** `*(count[]);`
type `(*name)[];`	Pointer to an (`open`) array of **type**	**int** `(*count) [];`
type `&name;`	Reference to **type** (C++ only)	**int** `&count;`
type `name();`	Function returning ***type***	**int** `count();`
type `*name();`	Function returning pointer to ***type***	**int** `*count();`
type `*(name());`	Same as above	**int** `*(count());`
type `(*name)();`	Pointer to function returning ***type***	**int** `(*count) ();`

Note the need for parentheses in (**name*)[ ] *and* (**name*)( ); this is because the precedence of both the array declarator [ ] and the function declarator ( ) is higher than the pointer declarator *. The parentheses in *(*name*[ ]) are optional.

Note: See C++Builderdeclaration syntax for the declarator syntax. The definition covers both identifier and function declarators.

Storage Class Specifiers

Storage classes specifiers are also called *type specifiers*. They dictate the location (data segment, register, heap, or stack) of an object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the declaration syntax, by its placement in the source code, or by both of these factors.

The keyword **mutable** does not affect the lifetime of the class member to which it is applied.

The storage class specifiers in C++Builder are:

auto	register
__declspec	static
extern	typedef
mutable	

Variable modifiers

See also

In addition to the storage class specifier keywords, a declaration can use certain *modifiers* to alter some aspect of the identifier. The modifiers available with C++Builder are summarized in C++Builder modifiers.

Mixed-language calling conventions

C++Builder allows your programs to easily call routines written in other languages, and vice versa. When you mix languages , you have to deal with two important issues: identifiers and parameter passing.

By default, C++Builder saves all global identifiers in their original case (lower, upper, or mixed) with an underscore "_" prepended to the front of the identifier. To remove the default, you can use the **-u** command-line option.

Note: The section Linkage tells how to use **extern**, which allows C names to be referenced from a C++ program.

Calling conventions summarizes the effects of a modifier applied to a called function. For every modifier, the table shows the order in which the function parameters are pushed on the stack. Next, the table shows whether the calling program (the *caller*) or the called function (the *callee*) is responsible for popping the parameters off the stack. Finally, the table shows the effect on the name of a global function.

Calling conventions

Modifier	Push parameters	Pop parameters	Name change
_ _cdecl1	Right first	Caller	'_' prepended
_ _fastcall	Left first	Callee	'@' prepended
_ _pascal	Left first	Callee	Uppercase
_ _stdcall	Right first	Callee	No change

1. This is the default.

Note: __fastcall and **_ _stdcall** are subject to name mangling. See the description of the -VC option.

const

Syntax

```
const <variable name> [ = <value> ] ;
<function name> ( const <type>*<variable name> ;)
<function name> const;
```

Description

Use the **const** modifier to make a variable value unmodifiable.

Use the **const** modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a **const** result in a compiler error.

A **const** pointer cannot be modified, though the object to which it points can be changed. Consider the following examples.

```
   const float pi   = 3.14;
   const  maxint  = 12345;   // When used by itself, const is equivalent to
  int.
   char *const str1 = "Hello, world";           // A constant pointer
   char const *str2 = "Borland International";  // A pointer to a constant
  character string.
```

Given these declarations, the following statements are illegal.

```
pi   = 3.0;          // Assigns a value to a const.
i     = maxint++;    // Increments a const.
str1 = "Hi, there!" // Points str1 to something else.
```

Using the const Keyword in C++ Programs

C++ extends **const** to include classes and member functions. In a C++ class definition, use the **const** modifier following a member function declaration. The member function is prevented from modifying any data in the class.

A class object defined with the **const** keyword attempts to use only member functions that are also defined with **const**. If you call a member function that is not defined as **const**, the compiler issues a warning that the a non-const function is being called for a **const** object. Using the **const** keyword in this manner is a safety feature of C++.

Warning: A pointer can indirectly modify a **const** variable, as in the following:

*(int *)&my_age = 35;

If you use the **const** modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int printf (const char *format, ...);
```

printf is prevented from modifying the format string.

volatile

See also Keywords

Syntax

```
volatile <data definition> ;
```

Description

Use the **volatile** modifier to indicate that a variable can be changed by a background routine, an interrupt routine, or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment. It also prevents the compiler from making the variable a register variable

```
volatile int ticks;
void timer( ) {
   ticks++;
}
void wait (int interval) {
   ticks = 0;
   while (ticks < interval);  // Do nothing
}
```

The routines in this example (assuming *timer* has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop since the loop doesn't change the value of *ticks*.

Note: C++ extends **volatile** to include classes and member functions. If you've declared a **volatile** object, you can use only its **volatile** member functions.

cdecl, _cdecl,   _ _cdecl

Example   Keywords

Syntax

```
cdecl <data/function definition> ;
_cdecl <data/function definition> ;
__cdecl <data/function definition> ;
```

Description

Use a **cdecl**, **_cdecl**, or **_ _cdecl** modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use **cdecl**, **_cdecl**, or **_ _cdecl** in front of a function, it effects how the parameters are passed (last parameter is pushed first, and the caller cleans up the stack). The **_ _cdecl** modifier overrides the compiler directives and IDE options.

The **cdecl**, **_cdecl**, and **__cdecl** keywords are specific to C++Builder.

pascal, _pascal, __pascal

Keywords

Syntax

```
pascal <data-definition/function-definition> ;
_pascal <data-definition/function-definition> ;
__pascal <data-definition/function-definition> ;
```

Description

Use the **pascal**, **_pascal**, and **__pascal** keywords to declare a variable or a function using a Pascal-style naming convention (the name is in uppercase).

In addition, **pascal** declares Pascal-style parameter-passing conventions when applied to a function header (first parameter pushed first; the called function cleans up the stack).

In C++ programs, functions declared with the **pascal** modifer will still be mangled.

_stdcall, __stdcall

Keywords

Syntax

```
__stdcall <function-name>
_stdcall <function-name>
```

Description

The **_stdcall** and **__stdcall** keywords force the compiler to generate function calls using the Standard calling convention. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. Such functions comply with the standard WIN32 argument-passing convention.

Note: The **__stdcall** modifier is subject to name mangling. See the description of the -VC option.

_fastcall, _ _fastcall

See also Keywords

Syntax

```
return-value _fastcall function-name(parm-list)
return-value __fastcall function-name(parm-list)
```

Description

Use the **__fastcall** modifier to declare functions that expect parameters to be passed in registers. The first three parameters are passed (from left to right) in EAX, EBX, and EDX, if they fit in the register. The registers are not used if the parameter is a floating-point or struct type.

All form class member functions must use the **__fastcall** convention.

The compiler treats this calling convention as a new language specifier, along the lines of _cdecl and _pascal

Functions declared using **_cdecl** or **_pascal** cannot also have the **_fastcall** modifiers because they use the stack to pass parameters. Likewise, the **__fastcall** modifier cannot be used together with _export.

The compiler prefixes the **_ _fastcall** function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

Note: The **__fastcall** modifier is subject to name mangling. See the description of the -VC option.

Multithread variables

Keywords

The keyword **_ _thread** is used in multithread programs to preserve a unique copy of global and static class variables. Each program thread maintains a private copy of a **_ _thread** variable for each threaded process.

The syntax is *Type* **__thread** *variable__name*. For example

```
 int __thread x;
```

declares an integer type variable that will be global but private to each thread in the program in which the statement occurs.

The **_ _thread** modifier can be used with global (file-scope) and static variables. The modifier cannot be used with pointers or functions. (However, you can have pointers to **_ _thread** objects.) A program element that requires run-time initialization or run-time finalization cannot be declared to be a **_ _thread** type. The following declarations require run-time initialization and are therefore illegal.

```
int f( );
int __thread x = f( );   // illegal
```

Instantiation of a class with a user-defined constructor or destructor requires run-time initialization and is therefore illegal.

```
class X  {
   X( );
   ~X( );
};
X __thread myclass;   // illegal
```

Function modifiers

See also

This section presents descriptions of the C++Builder function modifiers

You can use the **_ _export** and **_ _import** modifiers to modify functions.

In 32-bit programs the keyword can be applied to class, function, and variable declarations

The **_ _export** modifier makes the function exportable from Windows. The **_ _import** modifier makes a function available to a Windows program. The keywords are used in an executable (if you don't use smart callbacks) or in a DLL.

Functions declared with the **_ _fastcall** modifier have different names than their non-**_ _fastcall** counterparts. The compiler prefixes the **_ _fastcall** function name with an @. This prefix applies to both unmangled C function names and to mangled C++ function names.

C++Builder modifiers

Modifier	Use with	Description
const1	Variables	Prevents changes to object.
volatile1	Variables	Prevents register allocation and some optimization. Warns compiler that object might be subject to outside change during evaluation.
_ _cdecl2	Functions	Forces C argument-passing convention. Affects linker and link-time names.
_ _cdecl2	Variables	Forces global identifier case-sensitivity and leading underscores.
_ _pascal	Functions	Forces Pascal argument-passing convention. Affects linker and link-time names.
_ _pascal	Variables	Forces global identifier case-insensitivity with no leading underscores.
_ _export	Functions/classes	Tells the compiler which functions or classes to export.
_ _import	Functions/classes	Tells the compiler which functions or classes to import.
_ _fastcall	Functions	Forces register parameter passing convention. Affects the linker and link-time names.
_ _stdcall	Function	Forces the standard WIN32 argument-passing convention.

1. C++ extends **const** and **volatile** to include classes and member functions.
2. This is the default.

Pointers

See also

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain C++Builder operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Note: See Referencing for a discussion of referencing and dereferencing.

Pointers to objects

See also

A pointer of type "pointer to object of ***type***" holds the address of (that is, points to) an object of ***type***. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

Pointers to functions

See also

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions.

A pointer to a function has a type called "pointer to function returning ***type***," where type is the function's return ***type***. For example,

```
void (*func)();
```

In C++, this is a pointer to a function taking no arguments, and returning **void**. In C, it's a pointer to a function taking an unspecified number of arguments and returning **void**. In this example,

```
void (*func)(int);
```

func* is a pointer to a function taking an **int argument and returning **void**.

For C++, such a pointer can be used to access static member functions. Pointers to class members must use pointer-to-member operators. See static_cast for details.

Pointer declarations

See also

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. C++Builder lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to void. And in C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer. See void for details.

Warning: You need to initialize pointers before using them.

If ***type*** is any predefined or user-defined type, including **void**, the declaration

```
type *ptr;   /* Uninitialized pointer */
```

declares *ptr* to be of type "pointer to ***type***." All the scoping, duration, and visibility rules apply to the *ptr* object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic NULL (defined in the standard library header files, such as stdio.h) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.

The pointer type "pointer to **void**" must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that *vptr* is a generic pointer capable of being assigned to by any "pointer to ***type***" value, including null, without complaint. Assignments without proper casting between a "pointer to ***type1***" and a "pointer to ***type2***," where ***type1*** and ***type2*** are different types, can invoke a compiler warning or error. If ***type1*** is a function and ***type2*** isn't (or vice versa), pointer assignments are illegal. If ***type1*** is a pointer to **void**, no cast is needed. Under C, if ***type2*** is a pointer to **void**, no cast is needed.

Pointer constants

See also

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be have its value changed. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```
int i;                        // i is an int
int * pi;                     // pi is a pointer to int (uninitialized)
int * const cp = &i;          // cp is a constant pointer to int
const int ci = 7;             // ci is a constant int
const int * pci;              // pci is a pointer to constant int
const int * const cpc = &ci;  // cpc is a constant pointer to a
                              // constant int
```

The following assignments are legal:

```
i = ci;                      // Assign const-int to int
*cp = ci;                    // Assign const-int to
                             // object-pointed-at-by-a-const-pointer
++pci;                       // Increment a pointer-to-const
pci = cpc;                   // Assign a const-pointer-to-a-const to a
                             // pointer-to-const
```

The following assignments are illegal:

```
ci = 0;                      // NO--cannot assign to a const-int
ci--;                        // NO--cannot change a const-int
*pci = 3;                    // NO--cannot assign to an object
                             // pointed at by pointer-to-const
cp = &ci;                    // NO--cannot assign to a const-pointer,
                             // even if value would be unchanged
cpc++;                       // NO--cannot change const-pointer
pi = pci;                    // NO--if this assignment were allowed,
                             // you would be able to assign to *pci
                             // (a const value) by assigning to *pi.
```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

Pointer arithmetic

See also

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type "pointer to ***type***" automatically take into account the size of ***type***; that is, the number of bytes needed to store a ***type*** object.

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to ***type***, adding an integral value to the pointer advances the pointer by that number of objects of ***type***. If ***type*** has size 10 bytes, then adding an integer 5 to a pointer to ***type*** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if *ptr1* points to the third element of an array, and *ptr2* points to the tenth element, then the result of `ptr2 - ptr1` would be 7.

The difference between two pointers has meaning only if both pointers point into the same array

When an integral value is added to or subtracted from a "pointer to ***type***," the result is also of type "pointer to ***type***."

There is no such element as "one past the last element," of course, but a pointer is allowed to assume such a value. If P points to the last array element, P + 1 is legal, but
P + 2 is undefined. If P points to one past the last array element, P - 1 is legal, giving a pointer to the last element. However, applying the indirection operator * to a "pointer to one past the last element" leads to undefined behavior.

Informally, you can think of P + *n* as advancing the pointer by (*n* * **sizeof**(***type***)) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type *ptrdiff_t* defined in stddef.h. This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of *ptrdiff_t*. In the expression *P1 - P2*, where *P1* and *P2* are of type pointer to type (or pointer to qualified type), *P1* and *P2* must point to existing elements or to one past the last element. If *P1* points to the *i*-th element, and *P2* points to the *j*-th element, *P1 - P2* has the value (*i - j*).

Pointer conversions

See also

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (***type****) will convert a pointer to type "pointer to ***type***."
See C++ specific for a discussion of C++ typecast mechanisms.

C++ reference declarations

See also

C++ reference types are closely related to pointer types. *Reference types* create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by *value*. In C++ you can pass arguments by value or by reference. See Referencing for complete details.

Arrays

See also

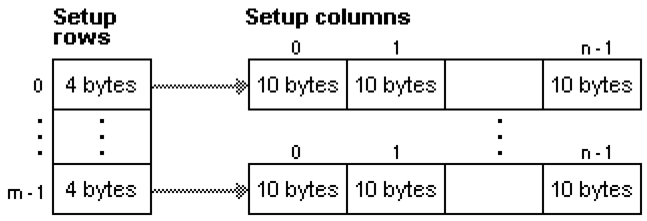
The declaration

type declarator [<constant-expression>]

declares an array composed of elements of ***type***. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. The following example shows one way to declare a two-dimensional array. The implementation is for three rows and five columns but it can be very easily modified to accept run-time user input.



```
/* DYNAMIC MEMORY ALLOCATION FOR A MULTIDIMENSIONAL OBJECT. */
#include <stdio.h>
#include <stdlib.h>

typedef long double TYPE;
typedef TYPE *OBJECT;
unsigned int rows = 3, columns = 5;

void de_allocate(OBJECT);

int main(void) {
  OBJECT matrix;
  unsigned int i, j;

  /* STEP 1: SET UP THE ROWS. */
  matrix = (OBJECT) calloc( rows, sizeof(TYPE *));

  /* STEP 2: SET UP THE COLUMNS. */
  for (i = 0; i < rows; ++i)
    matrix[i] = (TYPE *) calloc( columns, sizeof(TYPE));

    for (i = 0; i < rows; i++)
      for (j = 0; j < columns; j++)
        matrix[i][j] = i + j;     /* INITIALIZE */

  for (i = 0; i < rows; ++i) {
    printf("\n\n");
    for (j = 0; j < columns; ++j)
      printf("%5.2Lf", matrix[i][j]);
  de_allocate(matrix);
  return 0;
  }

void de_allocate(OBJECT x) {
  int i;
  for (i = 0; i < rows; i++)     /* STEP 1: DELETE THE COLUMNS */
    free(x[i]);

  free(x);    /* STEP 2: DELETE THE ROWS. */
  }
```

This code produces the following output:

```
0.00 1.00 2.00 3.00 4.00
1.00 2.00 3.00 4.00 5.00
2.00 3.00 4.00 5.00 6.00
```

In certain contexts, the first array declarator of a series might have no expression inside the brackets. Such an array is of indeterminate size. This is legitimate in contexts where the size of the array is not needed to reserve space.

For example, an **extern** declaration of an array object does not need the exact dimension of the array; neither does an array function parameter. As a special extension to ANSI C, C++Builder also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a pointer to the first element of the array.

Functions

See also

Functions are central to C and C++ programming. Languages such as Pascal distinguish between procedure and function. For C and C++, functions play both roles.

Member functions are sometimes referred to as *methods*.

Declarations and definitions

See also

Each program must have a single external function named *main* marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. They can be restricted by using the **static** storage class specifier (see Linkage).

Functions are defined in your source files or made available by linking precompiled libraries.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide C++Builder with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Note: In C++ you must always use function prototypes. We recommend that you also use them in C.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a definition and a declaration is that the definition has a function body.

Declarations and prototypes

See also

In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows

<type> *func*()

where ***type*** is the optional return type defaulting to **int**. In C++, this declaration means <***type***> *func*(**void**). A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

This problem was eased by the introduction of function prototypes with the following declaration syntax:

<type> *func(parameter-declarator-list);*

Note: You can enable a warning with the command-line compiler: "`Function called without a prototype.`"

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */
foo()
{
   int limit = 32;
   char ch = 'A';
   long mval;
   mval = lmax(limit,ch);     /* function call */
}
```

Since it has the function prototype for *lmax*, this program converts *limit* and *ch* to **long**, using the standard rules of assignment, before it places them on the stack for the call to *lmax*. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to *lmax* would not match in size or content what *lmax* was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function *strcpy* takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```

In C++, *func*() also declares a function taking no arguments

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as *printf*), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Note: stdarg.h and varargs.h contain macros that you can use in user-defined functions with variable numbers of parameters.

Here are some more examples of function declarators and prototypes:

```
int  f();   /* In C, a function returning an int with
             no information about parameters.
            This is the K&R "classic style." */

int f();    /* In C++, a function taking no arguments */

int  f(void);     /* A function returning an int that takes
                        no parameters. */

int  p(int,long); /* A function returning an int that
            accepts two parameters: the first,
             an int; the second, a long. */

int  _ _pascal q(void);      /* A pascal function returning
            an int that takes no parameters at all. */

int  printf(char *format,...;  /*   A function returning an int and
                  accepting a pointer to a char fixed
                  parameter and any number of additional
                  parameters of unknown type. */

int  (*fp)(int)        /* A pointer to a function returning an int
                  and requiring an int parameter. */
```

Definitions

See also

External function definitions gives the general syntax for external function definitions.

External function definitions

file

external-definition

file   external-definition

external-definition:

function-definition

declaration

asm-statement

function-definition:

<declaration-specifiers> declarator <declaration-list>

compound-statement

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):

1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
2. A return type, possibly **void**. The default is **int**.
3. Optional modifiers: **_ _pascal**, **_ _cdecl**, **_ _export**. The defaults depend on the compiler option settings.
4. The name of the function.
5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is `func(void)`. The old style of *func* is legal in C but antiquated and possibly unsafe.
6. A function body representing the code to be executed when the function is called.

Note: You can mix elements from 1 and 2.

Formal parameter declarations

See also

The formal parameter declaration list follows a syntax similar to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) {                   // no args
int func(T1 t1, T2 t2, T3 t3=1) {  // three simple parameters, one
                                   // with default argument
int func(T1* ptr1, T2& tref) {     // A pointer and a reference arg
int func(register int i) {         // Request register for arg
int func(char *str,...) {          /* One string arg with a variable number
  of other
                               args, or with a fixed number of args with var
  ying types */
```

In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, or enumerations; pointers or references to structures and unions; or pointers to functions or classes.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all have automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal parameter declarators

Function calls and argument conversions

See also

A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters.

Here is a summary of the rules governing how C++Builder deals with language modifiers and formal parameters in function calls, both with and without prototypes:

- The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
- A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, C++Builder converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard arithmetic conversions. When a function prototype is in scope, C++Builder converts the given argument to the type of the declared parameter as if by assignment

When a function prototype includes an ellipsis (...), C++Builder converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need to be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If your function prototype does not match the actual function definition, C++Builder will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught.
C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Structures

See also

A *structure* is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The C++Builder structure type lets you handle complex data structures almost as easily as single variables. Structure initialization is discussed in Arrays, structures, and unions.

In C++, a structure type is treated as a class type with certain differences: default access is public, and the default for the base class is also public. This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example

```
struct mystruct { ... }; // mystruct is the structure tag
     .
     .
     .
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

Untagged structures and typedefs

See also

If you omit the structure tag, you can get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited *struct-id-list* to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];       // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, arry[20];
```

Usually, you don't need both a tag and a **typedef**: either can be used in structure declarations.

Untagged structure and union members are ignored during initialization.

Structure member declarations

See also

The *member-decl-list* within the braces declares the types and names of the structure members using the declarator syntax shown in C++Builder declaration syntax.

A structure member can be of any type, with two exceptions

- The member type cannot be the same as the **struct** type being currently declared:

  ```
  struct mystruct { mystruct s } s1, s2; // illegal
  ```

  However, a member can be a pointer to the structure being declared, as in the following example:

  ```
  struct mystruct { mystruct *ps } s1, s2; // OK
  ```

  Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.
- Except in C++, a member cannot have the type "function returning...," but the type "pointer to function returning..." is allowed. In C++, a **struct** can have member functions.

Note: You can omit the **struct** keyword in C++.

Structures and functions

See also

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void);  // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s);         // directly
void func2(mystruct *sptr);     // via a pointer
void func3(mystruct &sref);     // as a reference (C++ only)
```

Structure member access

See also

Structure and union members are accessed using the following two selection operators:

- . (period)
- -> (right arrow)

Suppose that the object *s* is of struct type *S*, and *sptr* is a pointer to *S*. Then if *m* is a member identifier of type *M* declared in *S*, the expressions *s.m* and *sptr->m* are of type *M*, and both represent the member object *m* in *S*. The expression *sptr->m* is a convenient synonym for `(*sptr).m`.

The operator . is called the direct member selector and the operator -> is called the indirect (or pointer) member selector. For example:

```
struct mystruct
{
   int i;
   char str[21];
   double d;
} s, *sptr = &s;
    .
    .
    .
s.i = 3;            // assign to the i member of mystruct s
sptr -> d = 1.23;   // assign to the d member of mystruct s
```

The expression *s.m* is an lvalue, provided that s is an lvalue and *m* is not an array type. The expression *sptr->m* is an lvalue unless *m* is an array type.

If structure *B* contains a field whose type is structure *A*, the members of *A* can be accessed by two applications of the member selectors

```
struct A {
   int j;
   double x;
};
struct B {
   int i;
   struct A a;
   double d;
} s, *sptr;
    .
    .
    .
s.i = 3;            // assign to the i member of B
s.a.j = 2;          // assign to the j member of A
sptr->d = 1.23;     // assign to the d member of B
(sptr->a).x = 3.14  // assign to x member of A
```

Each structure declaration introduces a unique structure type, so that in

```
struct A {
   int i,j;
   double d;
} a, a1;
struct B {
   int i,j;
   double d;
} b;
```

the objects *a* and *a1* are both of type struct *A*, but the objects *a* and *b* are of different structure types. Structures can be assigned only if the source and destination have the same type:

```
a = a1;   // OK: same type, so member by member assignment
a = b;    // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign member-by-member */
```

Structure word alignment

See also

Memory is allocated to a structure member-by-member from left to right, from low to high memory address. In this example,

```
struct mystruct {
   int i;
   char str[21];
   double d;
} s;
```

the object s occupies sufficient memory to hold a 4-byte integer for a 32-bit program, a 21-byte string, and an 8-byte **double**. The format of this object in memory is determined by selecting the word alignment option. Without word alignment, s will be allocated 33 contiguous bytes by the 32-bit compiler.

Word alignment is off by default. If you turn on word alignment, C++Builder pads the structure with bytes to ensure the structure is aligned as follows:

32-bit compiler alignment

1 The structure boundaries are defined by 4-byte multiples.

2 For any non-**char** member, the offset will be a multiple of the member size. A **short** will be at an offset that is some multiple of 2 **ints** from the start of the structure.

3 One to three bytes can be added (if necessary) at the end to ensure that the whole structure contains a 4-byte multiple.

For the 32-bit compiler, with word alignment on, three bytes would be added before the **double**, making a 36-byte object.

Structure name spaces

See also

Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example

```
goto s;
    .
    .
    .
s:           // Label
struct s {   // OK: tag and label name spaces different
   int s;    // OK: label, tag and member name spaces different
   float s;  // ILLEGAL: member name duplicated
} s;         // OK: var name space different. In C++, this can only
             // be done if s does not have a constructor.
union s {    // ILLEGAL: tag space duplicate
   int s;    // OK: new member space
   float f;
} f;         // OK: var name space
struct t {
   int s;    // OK: different member space
    .
    .
    .
} s;         // ILLEGAL: var name duplicate
```

Incomplete declarations

See also

Incomplete declarations are also known as *forward* declarations.

A pointer to a structure type *A* can legally appear in the declaration of another structure *B* before *A* has been declared:

```
struct A;                  // incomplete
struct B { struct A *pa };
struct A { struct B *pb };
```

The first appearance of *A* is called *incomplete* because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of *B* doesn't need the size of *A*.

Bit fields

See also

When you write an application for a 16-bit platform, you can declare **signed** or **unsigned** integer members as bit fields from 1 to 16 bits wide. For 32-bit platforms a bit field can be as much as 32 bits wide. You specify the bit-field width and optional identifier as follows:

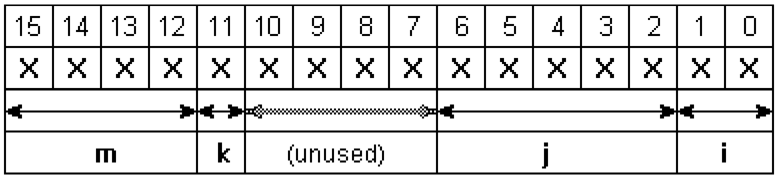
```
  type-specifier <bitfield-id> : width;
```

where *type-specifier* is **char**, **unsigned char**, **int**, or **unsigned int**. Bit fields are allocated from low-order to high-order bits within a word. The expression *width* must be present and must evaluate to a constant integer in the range 1 to 32, depending on the target platform.

If the bit field identifier is omitted, the number of bits specified in *width* is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused. For example:

```
struct mystruct
  int       i : 2;
  unsigned  j : 5;
  int         : 4;
  int       k : 1;
  unsigned  m : 4;
) a, b, c;
```

produces the following layout:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
m				k	(unused)				j					i	

Integer fields are stored in two's-complement form, with the leftmost bit being the MSB (most significant bit). With **int** (for example, **signed**) bit fields, the MSB is interpreted as a sign bit. A bit field of width 2 holding binary 11, therefore, would be interpreted as 3 if **unsigned**, but as -1 if **int**. In the previous example, the legal assignment `a.i = 6` would leave binary 10 = -2 in *a.i* with no warning. The signed **int** field *k* of width 1 can hold only the values -1 and 0, because the bit pattern 1 is interpreted as -1.

Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (. and ->) used for non-bit-field members. Also, bit fields pose several problems when writing portable code, since the organization of bits-within-bytes and bytes-within-words is machine dependent

The expression *&mystruct.x* is illegal if *x* is a bit field identifier, because there is no guarantee that *mystruct.x* lies at a byte address

Unions

See also

Union types are derived types sharing many of the syntactical and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
   int i;
   double d;
   char ch;
} mu, *muptr=&mu;
```

the identifier *mu*, of type **union** *myunion*, can be used to hold a 2-byte **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time

Note: Unions correspond to the variant record types of Pascal and Modula-2.

sizeof(**union** *myunion*) and **sizeof**(*mu*) both return 8, but 6 bytes are unused (padded) when *mu* holds an **int** object, and 7 bytes are unused when *mu* holds a **char**. You access union members with the structure member selectors (. and ->), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); //OK: displays mu.d = 4.016
  printf("mu.i = %d\n",mu.i); //peculiar result
  mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch); //OK: displays mu.ch = A
  printf("mu.d = %f\n",mu.d); //peculiar result
  muptr->i = 3;
printf("mu.i = %d\n",mu.i); //OK: displays mu.i = 3
```

The second *printf* is legal, since *mu.i* is an integer type. However, the bit pattern in *mu.i* corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Anonymous unions (C++ only)

See also

A union that doesn't have a tag and is not used to declare a named object (or other type) is called an *anonymous union*. It has the following form:

```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the `x.y` or `p->y` syntax.

Anonymous unions can't have member functions and at file level must be declared static. In other words, an anonymous union cannot have external linkage.

Union declarations

See also

The general declaration syntax for unions is similar to that for structures. The differences are

- Unions can contain bit fields, but only one can be active. They all start at the beginning of the union. (*Note that, because bit fields are machine dependent, they can pose problems when writing portable code.*)
- Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.
- Unions can be initialized only through their first declared member:

```
union local87 {
   int i;
   double d;
   } a = { 20 };
```

- A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union *can* have a constructor.

Enumerations

See also

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration,

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum days**, a variable *anyday* of this type, and a set of enumerators (*sun*, *mon*,...) with constant integer values

C++Builder is free to store enumerators in a single byte when `Treat enums as ints is unchecked` (O|C|Code Generation) or the **-b** flag is used. The default is on (meaning **enums** are always **ints**) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, that is the type of each enumerator

In C, a variable of an enumerated type can be assigned any value of type **int**--no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is,

```
anyday = mon;        // OK
anyday = 1;          // illegal, even though mon == 1
```

The identifier *days* is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```

In C++, you can omit the **enum** keyword if **days** is not the name of anything else in the same scope

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

The enumerators listed inside the braces are also known as *enumeration constants*. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (*sun*) is set to zero, and each succeeding enumerator is set to one more than its predecessor (*mon* = 1, *tues* = 2, and so on). See Enumeration constants for more on enumeration constants

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration,

```
/* Initializer expression can include previously declared enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
             quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, *nickel* the value 5, and *quarter* the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;         // OK
*daysptr = anyday;    // OK
mon = tues;           // ILLEGAL: mon is a constant
```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int mon = 11;
{
   enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
   /* enumerator mon hides outer declaration of int mon */
   struct days { int i, j;};   // ILLEGAL: days duplicate tag
   double sat;                 // ILLEGAL: redefinition of sat
}
mon = 12;                      // back in int mon scope
```

In C++, enumerators declared within a class are in the scope of that class.

In C++ it is possible to overload most operators for an enumeration. However, because the =, [ ], ( ), and -> operators must be overloaded as member functions, it is not possible to overload them for an **enum**. See the example on how to overload the postfix and prefix increment operators.

How to overload enum operators

```
// OVERLOAD THE POSTFIX AND PREFIX INCREMENT OPERATORS FOR enum
  #include <iostream.h>
  enum _SEASON { spring, summer, fall, winter };
  _SEASON operator++(_SEASON &s) {       // PREFIX INCREMENT
     _SEASON tmp = s; // SAVE THE ORIGINAL VALUE
     // DO MODULAR ARITHMETIC AND CAST THE RESULT TO _SEASON TYPE
     s = _SEASON( (s + 1) % 4 );         // INCREMENT THE ORIGINAL
     return s;                           // RETURN THE OLD VALUE
     }
  // UNNAMED int ARGUMENT IS NOT USED
  _SEASON operator++(_SEASON &s, int) { // POSTFIX INCREMENT
     _SEASON tmp = s;
     switch (s) {
        case spring: s = summer; break;
        case summer: s = fall; break;
        case fall:   s = winter; break;
        case winter: s = spring; break;
        }
     return (tmp);
     }
  int main(void) {
     _SEASON season = fall;
     cout << "\nThe season is " << season;
     cout << "\nIncrement the season: "<< ++season;
     cout << "\nNo change yet when using postfix: " << season++;
     cout << "\nFinally:" << season;
     return 0;
     }
```

This code produces the following output:

```
The season is 2
Increment the season: 3
No change yet when using postfix: 3
Finally:0
```

Assignment to enum types

The rules for expressions involving **enum** types have been made stricter. The compiler enforces these rules with error messages if the compiler switch **-A** is turned on (which means strict ANSI C++).

Assigning an integer to a variable of **enum** type results in an error:

```
enum color
{
  red, green, blue
};

int f()
{
  color c;
  c = 0;
  return c;
}
```

The same applies when passing an integer as a parameter to a function. Notice that the result type of the expression `flag1|flag2` is **int**:

```
enum e
{
  flag1 = 0x01,
  flag2 = 0x02
};

void p(e);

void f()
{
  p(flag1|flag2);
}
```

To make the example compile, the expression `flag1|flag2` must be cast to the **enum** type: `e (flag1|flag2)`.

Expressions

See also

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in C++Builder expressions, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Note: C++Builder expressions shows how identifiers and operators are combined to form grammatically legal "phrases."

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands.The standard conversions are detailed in Methods used in standard arithmetic conversions. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by C++Builder (see Evaluation order).

Expressions can produce an lvalue, an rvalue, or no value. Expressions might cause side effects whether they produce a value or not

The precedence and associativity of the operators are summarized in Associativity and precedence of C++Builder operators. The grammar in C++Builder expressions, completely defines the precedence and associativity of the operators

C++Builder expressions

primary-expression:

  literal

  this (C++ specific)

  :: *identifier* (C++ specific)

  :: *operator-function-name* (C++ specific)

  :: *qualified-name* (C++ specific)

  (*expression*)

  name

literal:

  integer-constant

  character-constant

  floating-constant

  string-literal

name:

  identifier

  operator-function-name (C++ specific)

  conversion-function-name (C++ specific)

  ~ *class-name* (C++ specific)

  qualified-name (C++ specific)

qualified-name: (C++ specific)

  qualified-class-name :: name

postfix-expression:

  primary-expression

  postfix-expression [*expression*]

postfix-expression (<*expression-list*>)

simple-type-name (<*expression-list*>) (C++ specific)

postfix-expression *.* *name*

postfix-expression *->* *name*

postfix-expression ++

postfix-expression --

const_cast < *type-id* > (*expression*) (C++ specific)

dynamic_cast < *type-id* > (*expression*) (C++ specific)

reinterpret_cast < *type-id* > (*expression*) (C++ specific)

static_cast < *type-id* > (*expression*) (C++ specific)

typeid (*expression*) (C++ specific)

typeid (*type-name*) (C++ specific)

expression-list:

assignment-expression

expression-list **,** *assignment-expression*

unary-expression:

postfix-expression

++ *unary-expression*

- - *unary-expression*

unary-operator *cast-expression*

sizeof *unary-expression*

sizeof (*type-name*)

allocation-expression (C++ specific)

deallocation-expression (C++ specific)

unary-operator: one of **& * + - !**

allocation-expression: (C++ specific)

<::> **new** <*placement*> *new-type-name* <*initializer*>

<::> **new** <*placement*> (*type-name*) <*initializer*>

placement: (C++ specific)

(*expression-list*)

new-type-name: (C++ specific)

type-specifiers <*new-declarator*>

new-declarator: (C++ specific)

ptr-operator <*new-declarator*>

new-declarator [<*expression*>]

deallocation-expression: (C++ specific)

<::> **delete** *cast-expression*

<::> **delete [ ]** *cast-expression*

cast-expression:

unary-expression

(type-name) *cast-expression*

pm-expression:

cast-expression

pm-expression **.*** *cast-expression* (C++ specific)

pm-expression **->*** *cast-expression* (C++ specific)

multiplicative-expression:

pm-expression

multiplicative-expression ***** *pm-expression*

multiplicative-expression **/** *pm-expression*

multiplicative-expression **%** *pm-expression*

additive-expression:

multiplicative-expression

additive-expression **+** *multiplicative-expression*

additive-expression **-** *multiplicative-expression*

shift-expression:

additive-expression

shift-expression **<<** *additive-expression*

shift-expression **>>** *additive-expression*

relational-expression:

shift-expression

relational-expression **<** *shift-expression*

relational-expression **>** *shift-expression*

relational-expression **<=** *shift-expression*

relational-expression **>=** *shift-expression*

equality-expression:

relational-expression

equality expression **==** *relational-expression*

equality expression **!=** *relational-expression*

AND-expression:

equality-expression

AND-expression **&** *equality-expression*

exclusive-OR-expression:

AND-expression

exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression:

exclusive-OR-expression

inclusive-OR-expression **|** *exclusive-OR-expression*

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression **&&** *inclusive-OR-expression* *logical-OR-expression:*

  logical-AND-expression

  logical-OR-expression **||** *logical-AND-expression*

conditional-expression:

  logical-OR-expression

  logical-OR-expression **?** *expression : conditional-expression*

assignment-expression:

  conditional-expression

  unary-expression   assignment-operator   assignment-expression

assignment-operator: one of

=   *=   /=   %=   +=   -=

<<   =>   >=   &=   ^= |=

expression:

  assignment-expression

  expression **,** *assignment-expression*

constant-expression:

  conditional-expression

Associativity and Precedence of Operators

Operators

There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator in the following table is indicated by its order in the table. The first category (on the first line) has the highest precedence. Operators on the same line have equal precedence.

Operators											Associativity
()	[]	->	::	.							left to right
!	~	+	-	++	--	&	*	sizeof	new	delete	right to left
.*	->*										left to right
*	/	%									left to right
+	-										left to right
<<	>>										left to right
<	<=	>	>=								left to right
==	!=										left to right
&											left to right
^											left to right
|											left to right
&&											left to right
||											right to left
?:											left to right
=	*=	/=	%=	+=	-=	&=	^=	|=	<<=	>>=	right to left
,											left to right

Expressions and C++

See also

C++ allows the overloading of certain standard C operators, as explained in Overloading Operator Functions. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator == might be defined in class *complex* to test the equality of two complex numbers without changing its normal usage with non-class data types.

An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the C language rules for operators and conversions might not apply to expressions in C++.

Evaluation order

See also

The order in which C++Builder evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. For example, consider the expression

```
i = v[i++];  // i is undefined
```

The value of i depends on whether i is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for sum and total. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value

C++Builder regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression

You can use parentheses to force the order of evaluation in expressions. For example, if you have the variables *a*, *b*, *c*, and *f*, then the expression *f* = *a* + (*b* + *c*) forces (*b* + *c*) to be evaluated before adding the result to *a*.

Errors and overflows

See also

Associativity and precedence of C++Builder operators. summarizes the precedence and associativity of the operators. During the evaluation of an expression, C++Builder can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2*n* arithmetic on *n*-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.See _matherr and signal.

Operators Summary

See also

Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

Arithmetic

Assignment

Bitwise

C++ specific

Comma

Conditional

Equality

Logical

Postfix Expression Operators

Primary Expression Operators

Preprocessor

Reference/Indirect operators

Relational

sizeof

typeid

All operators can be overloaded except the following:

`.`	C++ direct component selector
`.*`	C++ dereference
`::`	C++ scope access/resolution
`?:`	Conditional

Depending on context, the same operator can have more than one meaning. For example, the ampersand (&) can be interpreted as:

- a bitwise AND (A & B)
- an address operator (&A)
- in C++, a reference modifier

Note: No spaces are allowed in compound operators. Spaces change the meaning of the operator and will generate an error.

Primary Expression Operators

For ANSI C, the primary expressions are *literal* (also sometimes referred to as *constant*), identifier, and (*expression*). The C++ language extends this list of primary expressions to include the keyword **this**, scope resolution operator **::**, *name*, and the class destructor **~** (tilde).

The primary expressions are summarized in the following list.
primary-expression:
literal
this (C++ specific)
  :: *identifier* (C++ specific)
  :: *operator-function-name* (C++ specific)
  :: *qualified-name* (C++ specific)
  (*expression*)
  name
literal:
integer-constant
character-constant
  floating-constant
  string-literal
name:
identifier
operator-function-name (C++ specific)
  conversion-function-name (C++ specific)
  ~ *class-name* (C++ specific)
  qualified-name (C++ specific)
qualified-name: (C++ specific)
qualified-class-name **::** *name*

For a discussion of the primary expression **this**, see this (keyword). The keyword **this** cannot be used outside a class member function body.

The scope resolution operator allows reference to a type, object, function, or enumerator even though its identifier is hidden.

The parenthesis surrounding an *expression* do not change the unadorned expression itself.

The primary expression *name* is restricted to the category of primary expressions that sometimes appear after the member access operators **.** (dot) and **–>** . Therefore, name must be either an lvalue or a function. See also the discussion of member access operators.

An *identifier* is a primary expression, provided it has been suitably declared. The description and formal definition of identifiers is shown in Lexical Elements: Identifiers.

See the discussion on how to use the destructor operator ~ (tilde).

Postfix expression operators

See also Operators

Syntax

```
postfix-expression(<arg-expression-list>)
array declaration [constant-expression]
compound statement { statement list }
postfix-expression . identifier
postfix-expression -> identifier
```

Remarks

( )	use to group expressions, isolate conditional expressions, indicate function calls and function parameters
{ }	use as the start and end of compound statements
[ ]	use to indicate single and multidimensional array subscripts
.	use to access structure and union members
->	use to access structure and union members

The following postfix expressions let you make safe, explicit typecasts in a C++ program.

const_cast< T > (expression)

dynamic_cast< T > (expression)

reinterpret_cast< T > (expression)

static_cast< T > (expression)

To obtain runtime type identification (RTTI), use the **typeid()** operator. The syntax is as follows:

typeid(expression)

typeid(type-name)

Array subscript operator

See also Operators

Brackets ([ ]) indicate single and multidimensional array subscripts. The expression

```
<exp1>[exp2]
```

is defined as

```
*((exp1) + (exp2))
```

where either:

- exp1 is a pointer and exp2 is an integer or
- exp1 is an integer and exp2 is a pointer

Function call operator

See also   Operators

Syntax

```
postfix-expression(<arg-expression-list>)
```

Remarks

Parentheses ()

- group expressions
- isolate conditional expressions
- indicate function calls and function parameters

The value of the function call expression, if it has a value, is determined by the return statement in the function definition.

This is a call to the function given by the postfix expression.

arg-expression-list is a comma-delimited list of expressions of any type representing the actual (or real) function arguments.

Direct member selector

See also Example Operators

Syntax

```
postfix-expression . identifier
```

postfix-expression must be of type union or structure.

identifier must be the name of a member of that structure or union type.

Remarks

Use the selection operator (.) to access structure and union members.

Suppose that the object *s* is of struct type S and *sptr* is a pointer to S. Then, if *m* is a member identifier of type M declared in S, this expression:

```
s.m
```

are of type M, and represent the member object *m* in *s*.

Example

```
struct mystruct {
   int i
   char str[21]
   double d
} s, *sptr=&s
 ...
s.i = 3          // assign to the i member of mystruct s
```

The expression s.m is an lvalue, provided that s is not an lvalue and m is not an array type.

If structure B contains a field whose type is structure A, the members of A can be accessed by two applications of the member selectors.

Indirect member selector

See also Example Operators

Syntax

```
postfix-expression -> identifier
```

postfix-expression must be of type pointer to structure or pointer to union.

identifier must be the name of a member of that structure or union type.

The expression designates a member of a structure or union object. The value of the expression is the value of the selected member it will be an lvalue if and only if the postfix expression is an lvalue.

Remarks

You use the selection operator -> to access structure and union members.

Suppose that the object *s* is of struct type S and *sptr* is a pointer to S. Then, if *m* is a member identifier of type M declared in S, this expression:

```
sptr->m
```

is of type M, and represents the member object *m* in *s*.

The expression

```
s->sptr
```

is a convenient synonym for (*sptr).m.

-> Example

```
struct mystruct {
   int I;
   char str[21];
   double d;
} s, *sptr=&s;
  .
  .
  .
sptr->d = 1.23;   // assign to the d member of mystruct s
```

The expression sptr->m is an lvalue unless m is an array type.

If structure B contains a field whose type is structure A, the members of A can be accessed by two applications of the member selectors.

Increment/Decrement operators

Operators

Increment operator (++)

Syntax

```
postfix-expression ++         (postincrement)
++ unary-expression           (preincrement)
```

The expression is called the operand it must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue..

Postincrement operator

The value of the whole expression is the value of the postfix expression before the increment is applied.

After the postfix expression is evaluated,the operand is incremented by 1.

Preincrement operator

The operand is incremented by 1 before the expression is evaluated the value of the whole expression is the incremented value of the operand.

The increment value is appropriate to the type of the operand.

Pointer types follow the rules for pointer arithmetic.

Decrement operator (--)

Syntax

```
postfix-expression --         (postdecrement)
-- unary-expression           (predecrement)
```

The decrement operator follows the same rules as the increment operator, except that the operand is decremented by 1 after or before the whole expression is evaluated.

Unary operators

See also Operators

Syntax

```
<unary-operator> <unary expression>
```

OR

```
<unary-operator> <type><unary expression>
```

Remarks

Unary operators group right-to-left.

C++Builder provides the following unary operators:

! Logical negation

* Indirection

++ Increment

~ Bitwise complement

-- Decrement

- Unary minus

+ Unary plus

Reference/Dereference Operators

See also Operators

Syntax

```
& cast-expression
* cast-expression
```

Remarks

The **&** and ***** operators work together to reference and dereference pointers that are passed to functions.

Referencing operator (&)

Use the reference operator to pass the address of a pointer to a function outside of *main().*

The cast-expression operand must be one of the following:

- a function designator
- an lvalue designating an object that is not a bit field and is not declared with a register storage class specifier

If the operand is of type <*type*>, the result is of type pointer to <*type*>.

Some non-lvalue identifiers, such as function names and array names, are automatically converted into "pointer-to-X" types when they appear in certain contexts. The **&** operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following example:

```
T t1 = 1, t2 = 2;
T *ptr = &t1;       // Initialized pointer
*ptr = t2;            // Same effect as t1 = t2
```

`T *ptr = &t1` is treated as

```
T *ptr;
ptr = &t1;
```

So it is *ptr*, or **ptr*, that gets assigned. Once *ptr* has been initialized with the address *&t1*, it can be safely dereferenced to give the lvalue **ptr*.

Indirection operator (*)

Use the asterisk (*****) in a variable expression to create pointers. And use the indirect operator in external functions to get a pointer's value that was passed by reference.

If the operand is of type *pointer to function*, the result is a function designator.

If the operand is a pointer to an object, the result is an lvalue designating that object.

The result of indirection is undefined if either of the following occur:

1. The *cast-expression* is a null pointer.
2. The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Note: & can also be the bitwise AND operator.

* can also be the multiplication operator.

Plus and Minus Operators

See also Operators

Unary

In these unary + - expressions

```
+ cast-expression
- cast-expression
```

the cast-expression operand must be of arithmetic type.

Results

+ cast-expression	Value of the operand after any required integral promotions.
- cast-expression	Negative of the value of the operand after any required integral promotions.

Binary

Syntax

```
add-expression + multiplicative-expression
add-expression - multiplicative-expression
```

Legal operand types for op1 + op2:

1. Both op1 and op2 are of arithmetic type.
2. op1 is of integral type, and op2 is of pointer to object type.
3. op2 is of integral type, and op1 is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Legal operand types for op1 - op2:

1. Both op1 and op2 are of arithmetic type.
2. Both op1 and op2 are pointers to compatible object types.
3. op1 is of pointer to object type, and op2 is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Note: The unqualified type <type> is considered to be compatible with the qualified types const type, volatile type,and const volatile type.

Arithmetic Operators

See also   Operators

Syntax

```
+ cast-expression
- cast-expression
add-expression + multiplicative-expression
add-expression - multiplicative-expression
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
postfix-expression ++       (postincrement)
++ unary-expression         (preincrement)
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

Remarks

Use the arithmetic operators to perform mathematical computations.

The unary expressions of + and - assign a positive or negative value to the cast-expression.

+ (addition), - (subtraction), * (multiplication), and / (division) perform their basic algebraic arithmetic on all data types, integer and floating point.

% (modulus operator) returns the remainder of integer division and cannot be used with floating points.

++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.

-- (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Binary operators

See also Operators

These are the binary operators in C++Builder:

Arithmetic	+	Binary plus (add)
	-	Binary minus (subtract)
	*	Multiply
	/	Divide
	%	Remainder (modulus)
Bitwise	<<	Shift left
	>>	Shift right
	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
	|	Bitwise inclusive OR
Logical	&&	Logical AND
	||	Logical OR
Assignment	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	|=	Assign bitwise OR
Relational	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
	==	Equal to
	!=	Not equal to
Component selection	.	Direct component selector
	->	Indirect component selector
Class-member	::	Scope access/resolution
	.*	Dereference pointer to class member
	->*	Dereference pointer to class member
Conditional	? :	Actually a ternary operator for example,
	a ? x : y	"if a then x else y"
Comma	,	Evaluate

Multiplicative Operators

Operators

Syntax

```
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
```

Remarks

There are three multiplicative operators:

*	(multiplication)
/	(division)
%	(modulus or remainder)

The usual arithmetic conversions are made on the operands.

(op1 * op2)   Product of the two operands

(op1 / op2)   Quotient of (op1 divided by op2)

(op1 % op2)   Remainder of (op1 divided by op2)

For / and %, op2 must be nonzero op2 = 0 results in an error. (You can't divide by zero.)

When op1 and op2 are integers and the quotient is not an integer:

1. If op1 and op2 have the same sign, op1 / op2 is the largest integer less than the true quotient, and op1 % op2 has the sign of op1.
2. If op1 and op2 have opposite signs, op1 / op2 is the smallest integer greater than the true quotient, and op1 % op2 has the sign of op1.

Note: Rounding is always toward zero.

* is context sensitive and can be used as the pointer reference operator.

Bitwise operators

See also Operators

Syntax

```
AND-expression &  equality-expression
exclusive-OR-expr ^ AND-expression
inclusive-OR-expr exclusive-OR-expression
~cast-expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

Remarks

Use the bitwise operators to modify the individual bits rather than the number.

Operator	What it does
&	bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
|	bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
^	bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
~	bitwise complement; inverts each bit. ~ is used to create destructors.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and assigns the left most bit to 0.
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns the right most bit to 0.

Both operands in a bitwise expression must be of an integral type.

Bit value		Results of		
E1	**E2**	**E1 & E2**	**E1 ^ E2**	**E1 | E2**
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Note: &, >>, << are context sensitive. & can also be the pointer reference operator.

>> can also be the input operator in I/O expressions.

<< can also be the output operator in I/O expressions.

Relational Operators

See also Operators

Syntax

```
relational-expression  <   shift-expression
relational-expression  >   shift-expression
relational-expression  <=  shift-expression
relational-expression  >=  shift-expression
```

Remarks

Use relational operators to test equality or inequality of expressions. If the statement evaluates to be **true** it returns a nonzero character; otherwise it returns **false** (0).

>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

In the expression

```
E1 <operator> E2
```

the operands must follow one of these conditions:

1. Both E1 and E2 are of arithmetic type.
2. Both E1 and E2 are pointers to qualified or unqualified versions of compatible types.
3. One of E1 and E2 is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void.
4. One of E1 or E2 is a pointer and the other is a null pointer constant.

Equality operators

See also

There are two equality operators: **==** and **!=**. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

Note: Notice that **==** and **!=** have a lower precedence than the relational operators **<** and **>**, **<=**, and **>=**. Also, **==** and **!=** can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

```
equality-expression:
  relational-expression
  equality-expression == relational-expression
  equality-expression != relational-expression
```

Logical Operators

Operators

Syntax

```
logical-AND-expr  && inclusive-OR-expression
logical-OR-expr   || logical-AND-expression
! cast-expression
```

Remarks

Operands in a logical expression must be of scalar type.

&&	logical AND; returns **true** only if both expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **false**, the second expression is not evaluated.
||	logical OR; returns **true** if either of the expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **true**, the second expression is not evaluated.
!	logical negation; returns **true** if the entire expression evaluates to be nonzero, otherwise returns **false**. The expression !E is equivalent to (0 == E).

Conditional Operator

Operators

Syntax

```
logical-OR-expr ? expr : conditional-expr
```

Remarks

The conditional operator ?: is a ternary operator.

In the expression E1 ? E2 : E3, E1 evaluates first. If its value is **true**, then E2 evaluates and E3 is ignored. If E1 evaluates to **false**, then E3 evaluates and E2 is ignored.

The result of E1 ? E2 : E3 will be the value of either E2 or E3 depending upon which evaluates.

E1 must be a scalar expression. E2 and E3 must obey one of the following rules:

1. Both of arithmetic type. E2 and E3 are subject to the usual arithmetic conversions, which determines the resulting type.
2. Both of compatible **struct** or **union** types. The resulting type is the structure or union type of E2 and E3.
3. Both of **void** type. The resulting type is **void**.
4. Both of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
5. One operand is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
6. One operand is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**. The resulting type is that of the non-pointer-to-**void** operand.

Assignment Operators

See also Operators

Syntax

```
unary-expr  assignment-op  assignment-expr
```

Remarks

The assignment operators are:

```
=     *=      /=      %=     +=     -=
<<=   >>=     &=      ^=     |=
```

The = operator is the only simple assignment operator, the others are compound assignment operators.

In the expression E1 = E2, E1 must be a modifiable lvalue. The assignment expression itself is not an lvalue.

The expression

```
E1 op= E2
```

has the same effect as

```
E1 = E1 op E2
```

except the lvalue E1 is evaluated only once. For example, E1 += E2 is the same as E1 = E1 + E2.

The expression's value is E1 after the expression evaluates.

For both simple and compound assignment, the operands E1 and E2 must obey one of the following rules:

1. E1 is a qualified or unqualified arithmetic type and E2 is an arithmetic type.
2. E1 has a qualified or unqualified version of a structure or union type compatible with the type of E2.
3. E1 and E2 are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
4. Either E1 or E2 is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
5. E1 is a pointer and E2 is a null pointer constant.

Note: Spaces separating compound operators (+<space>=) will generate errors.

Comma Punctuator and Operator

See also Operators

Syntax

```
expression , assignment-expression
```

Remarks

The comma separates elements in a function argument list.

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

The left operand E1 is evaluated as a void expression, then E2 is evaluated to give the result and type of the comma expression. By recursion, the expression

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each Ei, with the value and type of En giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls `func` with three arguments `(i, 5, k)`, not four.

C++ Specific Operators

See also   Operators

The operators specific to C++ are:

`::`	Scope access (or resolution) operator
`.*`	Dereference pointers to class members
`->*`	Dereference pointers to pointers to class members
const_cast	adds or removes the **const** or **volatile** modifier from a type
delete	dynamically deallocates memory
dynamic_cast	converts a pointer to a desired type
new	dynamically allocates memory
reinterpret_cast	replaces casts for conversions that are unsafe or implementation dependent.
static_cast	converts a pointer to a desired type
typeid	gets run-time identification of types and expressions

Use the scope access (or resolution) operator ::(two semicolons) to access a global (or file duration) name even if it is hidden by a local redeclaration of that name.

Use the .* and ->* operators to dereference pointers to class members and pointers to pointers to class members.

Statements

See also

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. C++Builder statements shows the syntax for statements.

C++Builder statements

statement:

labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

asm-statement

declaration (C++ specific)

labeled-statement:

identifier : statement

case *constant-expression : statement*

default : *statement*

compound-statement:

{ <declaration-list> <statement-list> }

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

expression-statement:

<expression> ;

asm-statement:

asm *tokens newline*

asm tokens;

asm *{ tokens; <tokens;>= <tokens;>}*

selection-statement:

if *(expression) statement*

if *(expression) statement else statement*

switch *(expression) statement*

iteration-statement:

while *(expression) statement*

do *statement while (expression) ;*

for *(for-init-statement <expression> ; <expression>) statement* *for-init-statement:*

    expression-statement

    declaration (C++ specific)

jump-statement:

    goto *identifier ;*

    continue *;*

    break *;*

    return *<expression> ;*

Blocks

See also

A compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces (**{ }**). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth.

Labeled statements

See also

A statement can be labeled in two ways:

- *label-identifier* **:** *statement*

  The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.
- **case** *constant-expression* **:** *statement*

  default : *statement*

  Case and default labeled statements are used only in conjunction with switch statements.

Expression statements

See also

Any expression followed by a semicolon forms an *expression statement*:

```
<expression>;
```

C++Builder executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls

The *null statement* is a special case, consisting of a single semicolon (`;`). The null statement does nothing, and is therefore useful in situations where the C++Builder syntax expects a statement but your program does not need one.

Selection statements

See also

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

Iteration statements

See also

Iteration statements let you loop a set of statements. There are three forms of iteration in C++Builder: **while**, **do while**, and **for** loops.

Jump statements

See also

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**

C++ specifics

See also

C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can’t compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function *func* declared twice in C with different argument types invokes a duplicated name error. Under C++, however, *func* will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. This topic discusses the aspects of C++ that can be used independently of classes, then describes the specifics of classes and class mechanisms.

See C++ Exception Handling and C-Based Structured Exceptions for details on compiling C and C++ programs with exception handling.

Namespaces overview

See Also

Most nontrivial applications consist of more than one source file. The files can be authored and maintained by more than one developer. Eventually, the separate files are organized and linked to produce the final application. Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as function or class body, or translation unit) must share the same global namespace. Therefore, multiple definitions of names discovered while linking separate modules require some way to distinguish each name. The solution to the problem of name clashes in the global scope is provided by the C++ namespace mechanism.

The namespace mechanism allows an application to be partitioned into a number of subsystems. Each subsystem can define and operate within its own scope. Each developer is free to introduce whatever identifiers are convenient within a subsystem without worrying about whether such identifiers are being used by someone else. The subsystem scope is known throughout the application by a unique identifier.

It only takes two steps to use C++ namespaces. The first is to uniquely identify a namespace with the keyword **namespace**. The second is to access the elements of an identified namespace by applying the **using** keyword.

Defining a namespace

SeeAlso

The grammar for defining a namespace is

original-namespace-name:

identifier

namespace-definition:

original-namespace-definition

extension-namespace-definition

unnamed-namespace-definition

Grammatically, there are three ways to define a namespace with the **namespace** keyword:

original-namespace-definition:

namespace *identifier* { *namespace-body* }

extension-namespace-definition:

namespace *original-namespace-name* { *namespace-body* }

unnamed-namespace-definition:

namespace { *namespace-body* }

The body is an optional sequence of declarations. The grammar is

namespace-body:

declaration-seq opt

Example for extending namespaces

```
// An example for extending namespaces
#include <iostream.h>
   struct S { };
   class C  { };

   namespace ALPHA {  // ALPHA is an original identifier.
      void g(struct S) {
         cout << "Processing a structure argument" << endl;
         }
      }

   using ALPHA::g;  // using-declaration

   /*** After the using-declaration above, subsequent attempts
        to overload the g() function are ignored. ***/
   namespace ALPHA { // Extending the ALPHA namespace
      void g( C& ) { // Overloaded version of function
         cout << "Processing a class argument." << endl;
         }
      }

   int main() {
      S mystruct;
      C myclass;

      g(mystruct);

      // The following function call fails at compile-time
      // because there is no overloaded version for this case.
 //      g(myclass);
      return 0;
   }
```

Output:

```
Processing a structure argument
```

Declaring a namespace

SeeAlso

An original namespace declaration should use an identifier that has not been previously used as a global identifier.

```
namespace ALPHA {  /* ALPHA is the identifier of this namespace. */
  /* your program declarations */
  long double LD;
  float f(float y) { return y; }
  }
```

A namespace identifier must be known in all translation units where you intend to access it's elements.

Namespace alias

You can use an alternate name to refer to a namespace identifier. An alias is useful when you need to refer to a long, unwieldy namespace identifier.

```
namespace BORLAND_INTERNATIONAL {
   /* namespace-body */
   namespace NESTED_BORLAND_INTERNATIONAL {
      /* namespace-body */
      }
   }

// Alias namespace
namespace BI = BORLAND_INTERNATIONAL;

// Use access qualifier to alias a nested namespace
namespace NBI = BORLAND_INTERNATIONAL::NESTED_BORLAND_INTERNATIONAL;
```

Extending a namespace

Example

Namespaces are discontinuous and open for additional development. If you redeclare a namespace, the effect is that you extend the original namespace by adding new declarations. Any extensions that are made to a namespace after a using-declaration, will not be known at the point at which the using-declaration occurs. Therefore, all overloaded versions of some function should be included in the namespace before you declare the function to be in use.

Anonymous namespaces

Example

The C++ grammar allows you to define anonymous namespaces. To do this,you use the keyword **namespace** with no identifier before the enclosing brace.

```
namespace {           // Anonymous namespace
   // Declarations
   }
```

All anonymous, unnamed namespaces in global scope (that is, unnamed namespaces that are not nested) of the same translation unit share the same namespace. This way you can make static declarations without using the **static** keyword.

Each identifier that is enclosed within an unnamed namespace is unique within the translation unit in which the unnamed namespace is defined.

Example

In file ANON1.CPP

```
#include <iostream.h>
extern void func(void);

namespace {           // Anonymous
   float pi = 3.14;  // Unique identifier known only in this file
   }

void main() {
   float pi = 0.1;
   cout << "pi = " << pi << endl;
   func();
   }
```

In file ANON2.CPP

```
#include <iostream.h>

namespace {              // Anonymous namespace
   float pi = 10.0001;  // Unique identifier known only in this file
   void func(void) {
       cout << "First func() called; pi = " << pi;
       }
   }
   void func(void) {
       cout << "Second func() called; pi = " << pi;
       }
```

Program output:

```
pi = 0.1
func() called; pi = 10.0001
```

Accessing elements of a namespace

There are three ways to access the elements of a namespace: by explicit access qualification, the using-declaration, or the using-directive. Remember that no matter which namespace you add to your local scope, identifiers in global scope (global scope is just another namespace) are still accessible by using the scope resolution operator **::**.

Explicit access qualification

Using directive

Using declaration

Accessing namespaces in classes

Example

You cannot use a **using** *directive* inside a class. However, the **using** *declarative* is allowed and can be quite useful.

Example

```
// An example for accessing a namespace within a class.
// This allows us to overload a function which is a base class member.

#include <iostream.h>
   class A {
   public:
      void func(char ch) { cout << "char = " << ch << endl; }
   };

   class B : public A {
   public:
//      using namespace A;   // ERROR. The using directive isn't allowed
      void func(char *str) { cout << "string = " << str << endl; }

      // The using declarative
      using A::func;        // Overload B::func()
   };

   int main() {
      B b;

      b.func('c');  // Calls A::func()
      b.func("c");  // Calls B::func()
      return 0;
   }
```

Using directive

Example

If you want to use several (or all of) the members of a namespace, C++ provides an easy way to get access to the complete namespace. The using-directive specifies that all identifiers in a namespace are in scope at the point that the using-directive statement is made. The grammar for the using-directive is as follows.

using-directive:

using namespace **::** opt *nested-name-specifier* opt *namespace-name*;

The using-directive is transitive. That means that when you apply the using directive to a namespace that contains using directives within itself, you get access to those namespaces as well. For example, if you apply the using directive in your program, you also get namespaces *A*, *ONE*, and *TWO*.

```
namespace A {
   using namespace ONE;  // This has been defined previously
   using namespace TWO;  // This also has been defined previously
   }
```

The using-directive does not add any identifiers to your local scope. Therefore, an identifier defined in more than one namespace won't be a problem until you actually attempt to use it. Local scope declarations take precedence by hiding all other similar declarations.

Example

```
// AN EXAMPLE OF THE using DIRECTIVE
#include <iostream.h>
   namespace F {
      float x = 9;
      }
   namespace G {
      using namespace F;
      float y = 2.0;
         namespace INNER_G {
            float z = 10.01;
            }
      }

   int main() {
      using namespace G;  // THIS DIRECTIVE GIVES YOU EVERYTHING DECLARED IN
  "G"
      using namespace G::INNER_G;  // THIS DIRECTIVE GIVES YOU ONLY
  "INNER_G"
      float x = 19.1;     // LOCAL DECLARATION TAKES PRECEDENCE
      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
      cout << "z = " << z << endl;
      return 0;
      }
```

Output:

```
   x = 19.1
   y = 2
   z = 10.01
```

Using declaration

Example

You can access namespace members individually with the using-declaration syntax. When you make a using declaration, you add the declared identifier to the local namespace. The grammar is

using-declaration:

using :: *unqualified-identifier*;

Example

```
// An example of the using declaration.
// The function g() is defined in two different namespaces.
#include <iostream.h>

namespace ALPHA {  /* ALPHA is the name of this namespace. */
   float f(float y) { return y; }
   void g() { cout << "ALPHA version" << endl; }
   }
namespace BETA {  /* BETA is the name of this namespace. */
      void g() { cout << "BETA version" << endl; }
         }

void main(void) {
// The using declaration identifies the desired version of g().
         using ALPHA::f;  // Qualified declaration
         using BETA::g;   // Qualified declaration
         float x = 0;

         // Access qualifiers are no longer needed.
         x = f(2.1);
         g();
         }
```

Explicit access qualification

You can explicitly qualify each member of a namespace. To do so, you use the namespace identifier together with the **::** scope resolution operator followed by the member name. For example, to access a specific member of namespace *ALPHA*, you write:

```
ALPHA::LD;  // Access a variable
ALPHA::f;   // Access a function
```

Explicit access qualification can always be used to resolve ambiguity. No matter which namespace (except anonymous namespace) is being used in your subsystem, you can apply the scope resolution operator **::** to access identifiers in any namespace (including a namespace already being used in the local scope) or the global namespace. Therefore, any identifier in the application can be accessed with sufficient qualification.

New-stye typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

New-style typecasting

See also

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

const_cast

See also

Syntax

```
const_cast< T > (arg)
```

Description

Use the **const_cast** operator to add or remove the **const** or **volatile** modifier from a type.

In the statement, `const_cast< T > (arg)`, *T* and *arg* must be of the same type except for **const** and **volatile** modifiers. The cast is resolved at compile time. The result is of type *T*. Any number of const or volatile modifiers can be added or removed with a single **const_cast** expression.

A pointer to **const** can be converted to a pointer to non-**const** that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A **const** object or a reference to **const** cast results in a non-**const** object or reference that is otherwise an identical type.

The **const_cast** operator performs similar typecasts on the **volatile** modifier. A pointer to volatile object can be cast to a pointer to non-**volatile** object without otherwise changing the type of the object. The result is a pointer to the original object. A **volatile**-type object or a reference to **volatile**-type can be converted into an identical non-**volatile** type.

dynamic_cast

See also Example

In the expression, `dynamic_cast< T > (ptr)`, *T* must be a pointer or a reference to a defined class type or **void***. The argument *ptr* must be an expression that resolves to a pointer or reference.

If `T` is void* then *ptr* must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if *T* is a pointer and *ptr* is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If *T* is a reference and `ptr` is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at runtime.

If successful, `dynamic_cast< T > (ptr)` converts *ptr* to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the Bad_cast exception is thrown.

Note: Runtime type identification (RTTI) is required for **dynamic_cast**.

// dynamic_cast Example

```
// HOW TO MAKE DYNAMIC CASTS
// This program must be compiled with the -RT (Generate RTTI) option.
#include <iostream.h>
#include <typeinfo.h>

class Base1
{
   // In order for the RTTI mechanism to function correctly,
   // a base class must be polymorphic.
   virtual void f(void) { /* A virtual function makes the class polymorphic
  */ }
};

class Base2 { };
class Derived : public Base1, public Base2 { };

int main(void) {
   try {
      Derived d, *pd;
      Base1 *b1 = &d;

      // Perform a downcast from a Base1 to a Derived.
      if ((pd = dynamic_cast<Derived *>(b1)) != 0) {
           cout << "The resulting pointer is of type "
                << typeid(pd).name() << endl;
         }
      else throw Bad_cast();

      // Attempt cast across the hierarchy.  That is, cast from
      // the first base to the most derived class and then back
      // to another accessible base.
      Base2 *b2;
      if ((b2 = dynamic_cast<Base2 *>(b1)) != 0) {
          cout << "The resulting pointer is of type "
               << typeid(b2).name() << endl;
         }
      else throw Bad_cast();
      }
   catch (Bad_cast) {
      cout << "dynamic_cast failed" << endl;
      return 1;
      }
   catch (...) {
      cout << "Exception handling error." << endl;
      return 1;
      }

   return 0;
}
```

reinterpret_cast

See also Example

Syntax

```
reinterpret_cast< T > (arg)
```

Description

In the statement, `reinterpret_cast< T > (arg)`, `T` must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral `arg` can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

// reinterpret_cast Example

```
// Use reinterpret_cast<Type>(expr) to replace (Type)expr casts
// for conversions that are unsafe or implementation dependent.

void func(void *v) {
        // Cast from pointer type to integral type.
        int i = reinterpret_cast<int>(v);

  .
  .
  .
}

void main() {
   // Cast from an integral type to pointer type.
   func(reinterpret_cast<void *>(5));

   // Cast from a pointer to function of one type to
   // pointer to function of another type.
   typedef void (* PFV)();

   PFV pfunc = reinterpret_cast<PFV>(func);

   pfunc();
   }
```

static_cast

See also

Syntax

```
static_cast< T > (arg)
```

Description

In the statement, `static_cast< T > (arg)`, `T` must be a pointer, reference, arithmetic type, or enum type. The arg-type must match the T-type. Both `T` and `arg` must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using **static_cast** achieves exactly the same thing.

Integral types can be converted to **enum** types. A request to convert `arg` to a value that is not an element of **enum** is undefined.

The null pointer is converted to itself.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class X to a pointer to some class Y if X is a base class for Y. A static conversion can be made only under the following conditions:

- if an unambiguous conversion exists from Y to X
- if X is not a virtual base class

An object can be explicitly converted to a reference type X& if a pointer to that object can be explicitly converted to an X*. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.

An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.

A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When `T` is a reference the result of `static_cast< T > (arg)` is an lvalue. The result of a pointer or reference cast refers to the original expression.

Run-time type identification (RTTI) overview

See also

Run-time type identification (RTTI) lets you write portable code that can determine the actual type of a data object at run time even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. Use the dynamic_cast operator to make run-time casts.

The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with typeid operator, which determines the actual type of its argument and returns a reference to an object of type **const** *typeinfo*, which describes that type.

You can also use a type name as the argument to **typeid**, and **typeid** will return a reference to a **const** *typeinfo* object for that type. The class typeinfo provides an **operator==** and an **operator!=** that you can use to determine whether two objects are of the same type. Class *typeinfo* also provides a member function name that returns a pointer to a character string that holds the name of the type.

_ _rtti Example

```
/* HOW TO GET RUN-TIME TYPE INFORMATION FOR POLYMORPHIC CLASSES.*/
#include <iostream.h>
#include <typeinfo.h>

class __rtti Alpha {         /* Provide RTTI for this class and */
                              /* all classes derived from it */

    virtual void func() {};   /* A virtual function makes */
                              /* Alpha a polymorphic class. */
};

class B : public Alpha {};

int main(void) {
   B Binst;            // Instantiate class B
   B *Bptr;            // Declare a B-type pointer
   Bptr = &Binst;      // Initialize the pointer

   // THESE TESTS ARE DONE AT RUN TIME
   try {
      if (typeid( *Bptr ) == typeid( B ) )
          // Ask "WHAT IS THE TYPE FOR *Bptr?"
          cout << "Name is " << typeid( *Bptr).name();
      if (typeid( *Bptr ) != typeid( Alpha ) )
          cout << "\nPointer is not an Alpha-type.";
      return 0;
      }
   catch (Bad_typeid) {
      cout << "typeid() has failed.";
      return 1;
      }
   }
```

Program Output

```
Name is B
Pointer is not an Alpha-type.
```

The typeid operator

See also Example Keywords

Syntax

```
typeid( expression )
typeid( type-name )
```

Description

You can use **typeid** to get run-time identification of types and expressions. A call to **typeid** returns a reference to an object of type **const** typeinfo. The returned object represents the type of the **typeid** operand.

If the **typeid** operand is a dereferenced pointer or a reference to a polymorphic type, **typeid** returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, **typeid** returns an object that represents the static type.

You can use the **typeid** operator with fundamental data types as well as user-defined types.

If the **typeid** operand is a dereferenced NULL pointer, the *Bad_typeid* exception is thrown.

_ _rtti and the -RT option

See also Example Keywords

RTTI is enabled by default in C++Builder. You can use the **-RT** command-line option to disable it (**-RT-**) or to enable it (**-RT**). If RTTI is disabled, or if the argument to **typeid** is a pointer or a reference to a non-polymorphic class, typeid returns a reference to a **const** *typeinfo* object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic run-time type identification (where appropriate) by using the C++Builder keyword **_ _rtti** in the class definition.

When you use the **-RT-** compiler option, if any base class is declared **_ _rtti**, then all polymorphic base classes must also be declared **_ _rtti**.

```
struct __rtti S1 { virtual s1func(); };  /* Polymorphic */
struct __rtti S2 { virtual s2func(); };  /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism (by using the **-RT-** compiler option), RTTI might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations with -RT-, you should declare *X* with the **_ _rtti** modifier. Otherwise, switching the order of the base classes for the class *X* results in the compile-time error Can't inherit non-RTTI class from RTTI base '*S1*'.

```
struct _ _rtti S1 { virtual func(); };    /* Polymorphic class */
struct S2 { };                            /*  Non-polymorphic class */
struct _ _rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with **_ _rtti**. This makes it safe to mix the order and type of classes.

In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared **_ _rtti**.

```
struct _ _rtti S1 {  };   // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { };   // The order is not essential
```

Applying either **_ _rtti** or using the -RT compiler option will not make a static class into a polymorphic class.

-RT option and destructors

When **-xd** is enabled, a pointer to a class with a virtual destructor can't be deleted if that class is not compiled with **-RT**. The **-RT** and **-xd** options are on by default.

Example

```
class Alpha {
public:
   virtual ~Alpha( ) { }
};
void func( Alpha *Aptr ) {
   delete Aptr;         // Error.  Alpha is not a polymorphic class type
   }
```

Referencing

See also

While in C, you pass arguments only by value; in C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects and let you pass arguments to functions by reference. See the following topics for a discussion of referencing.

Simple references

Reference arguments

Reference/Indirect operators

Note: C++ specific pointer referencing and dereferencing is discussed in C++ specific operators.

Simple references

See also

The reference declarator can be used to declare references outside functions:

```
int  i  = 0;
int &ir = i;   // ir is an alias for i
ir = 2;        // same effect as i = 2
```

Note that type& var, type &var, and type & var are all equivalent.

This creates the lvalue *ir* as an alias for *i*, provided the initializer is the same type as the reference. Any operations on *ir* have precisely the same effect as operations on *i*. For example, `ir = 2` assigns 2 to *i*, and `&ir` returns the address of *i*.

Reference arguments

See also

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir);    // ir is type "reference to int"
     .
     .
     .
int sum=3;
func1(sum);              // sum passed by value
func2(&sum);             // sum passed by reference
```

The *sum* argument passed by reference can be changed directly by *func2*. On the other hand, *func1* gets a copy of the *sum* argument (passed by value), so sum itself cannot be altered by *func1*.

When an actual argument *x* is passed by value, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of x itself. Of course, the function can return a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The C method for changing *x* uses the actual argument `&x`, the address of *x*, rather than *x* itself. Although &x is passed by value, the function can access x through the copy of `&x` it receives. Even if the function does not need to change *x*, it is still useful (though subject to potentially dangerous side effects) to pass `&x`, especially if *x* is a large data structure. Passing *x* directly by value involves wasteful copying of the data structure.

Compare the three implementations of the function *treble*:

Implementation 1

```
int treble_1(int n)
{
   return 3 * n;
}
  .
  .
  .
int x, i = 4;
x = treble_1(i);         // x now = 12, i = 4
  .
  .
  .
```

Implementation 2

```
void treble_2(int* np)
{
   *np = (*np) * 3;
}
  .
  .
  .
treble_2(int& i);        // i now = 12
```

Implementation 3

```
void treble_3(int& n)    // n is a reference type
{
   n = 3 * n;
}
```

```
   .
   .
   .
treble_3(i);             // i now = 36
```

The formal argument declaration ***type***`&  t` (or equivalently, ***type***`& t`) establishes *t* as type "reference to *type*." So, when *treble_3* is called with the real argument *i*, *i* is used to initialize the formal reference argument *n*. *n* therefore acts as an alias for *i*, so `n = 3*n` also assigns *3 * i* to *i*.

If the initializer is a constant or an object of a different type than the reference type, creates a temporary object for which the reference acts as an alias:

```
int& ir = 6;   /* temporary int object created, aliased by ir, gets value 6
  */
float f;
int& ir2 = f;  /* creates temporary int object aliased by ir2; f converted
                  before assignment */
ir2 = 2.0      // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

Scope resolution operator ::

The scope access (or resolution) operator **::** (two colons) lets you access a global (or file duration) member name even if it is hidden by a local redeclaration of that name. You can use a global identifiers by prefixing it with the resolution operator. To access a nested member name by specifying the class name and using the resolution operator. Therefore, `Alpha::func( )` and `Beta::func( )` are two different functions.

operator new

See also Example Operators

Syntax

```
<::> new <placement> type-name <(initializer)>
<::> new <placement> (type-name) <(initializer)>
```

Description

The **new** operator offers dynamic storage allocation, similar but superior to the standard library function malloc. The **new** operator must always be supplied with a data type in place of *type-name*. Items surrounded by angle brackets are optional. The optional arguments can be as follows:

- **::** operator, invokes the global version of **new**.
- *placement* can be used to supply additional arguments to **new**. You can use this syntax only if you have have an overloaded version of **new** that matches the optional arguments. See the discussion of the placement syntax.
- *initializer*, if present is used to initialize the allocation. Arrays cannot be initialized by the allocation operator.

A request for non-array allocation uses the appropriate **operator new()** function. Any request for array allocation will call the appropriate **operator new[]()** function. The selection of the allocation operator is done as follows:

Allocation of arrays of *Type*:

1. Attempts to use a class-specific array allocator:

   Type::**operator new[]()**

2. If the class-specific array allocator is not defined, the global version is used:

   ::operator new[]()

Allocation of non-arrays of *Type*:

1. Attempts to used the class-specific allocator:

   Type::**operator new()**

2. If the class-specific array allocator is not defined, the global version is used:

   ::operator new()

Allocation of single objects (that are not class-type) which are not held in arrays:

1. Memory allocation for a non-array object is by using the **::operator new()**. Note that this allocation function is always used for the predefined types. It is possible to overload this global operator function. However, this is generally not advised.

Allocation of arrays:

1. Use the global allocation operator:

   ::operator new[] ()

Note: Arrays of classes require the default constructor.

new tries to create an object of type *Type* by allocating (if possible) sizeof(*Type*) bytes in free store (also called the heap). **new** calculates the size of *Type* without the need for an explicit **sizeof** operator. Further, the pointer returned is of the correct type, "pointer to *Type*," without the need for explicit casting. The storage duration of the **new** object is from the point of creation until the operator **delete** destroys it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the allocated memory. By default, an allocation failure (such as insufficient or fragmented heap memory) results in the predefined exception xalloc being thrown. Your program should always be prepared to catch the *xalloc* exception before trying to access the new object (unless you use a new-handler).

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

operator delete

See also Example Operators

Syntax

```
<::> delete <cast-expression>
<::> delete [ ] <cast-expression>
delete <array-name> [ ];
```

Description

The **delete** operator offers dynamic storage deallocation, deallocating a memory block allocated by a previous call to new. It is similar but superior to the standard library function free.

You should use the **delete** operator to remove all memory which has been allocated by the **new** operator. Failure to free memory can result in memory leaks.

Operator new placement syntax

Example

The *placement* syntax for **operator new( )** can be used only if you have overloaded the allocation operator with the appropriate arguments. You can use the *placement* syntax when you want to use and reuse a memory space which you set up once at the beginning of your program.

When you use the overloaded **operator new( )** to specify where you want an allocation to be placed, you are responsible for deleting the allocation. Because you call your version of the allocation operator, you cannot depend on the global **::operator delete( )** to do the cleanup.

To release memory, you make an explicit call on the destructor. This method for cleaning up memory should be used only in special situations and with great care. If you make an explicit call of a destructor before an object that has been constructed on the stack goes out of scope, the destructor will be called again when the stackframe is cleaned up.

Handling Errors for the new Operator

See also

By default, **new** throws the *xalloc* exception when a request for memory allocation cannot be satisfied.

You can define a function to be called if the **new** operator fails. To tell the **new** operator about the new-handler function, use set_new_handler and supply a pointer to the new-handler. If you want **new** to return null on failure, you must use `set_new_handler(0)` .

The Operator new With Arrays

Example

If Type is an array, the pointer returned by operator new[]() points to the first element of the array. When creating multidimensional arrays with new, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];     // OK
mat_ptr = new int[n][10][12];     // OK
mat_ptr = new int[3][][12];       // illegal
mat_ptr = new int[][10][12];      // illegal
```

Although the first array dimension can be a variable, all following dimensions must be constants.

The delete Operator with Arrays

Arrays are deleted by operator delete[](). You must use the syntax **delete []** *expr* when deleting an array.

```
char * p;

void func()
{
   p = new char[10];    //  allocate 10 chars
   delete[] p;          // delete 10 chars
}
```

C++ 2.0 code required the array size to be named in the delete expression. In order to allow 2.0 code to compile, C++Builder issues a warning and simply ignores any size that is specified. For example, if the preceding example reads delete[10] p and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

operator new

See also

By default, if there is no overloaded version of new, a request for dynamic memory allocation always uses the global version of new, **::operator new()**. A request for array allocation calls **::operator new[]()**. With class objects of type name, a specific operator called *name***::operator new()** or *name***::operator new[]()** can be defined. When **new** is applied to class name objects it invokes the appropriate *name***::operator new** if it is present; otherwise, the global **::operator new** is used.

Only the **operator new()** function will accept an optional initializer. The array allocator version, **operator new[](),** will not accept initializers. In the absence of explicit initializers, the object created by new contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the default constructor. The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

Overloading the operator new

Example

The global ::operator new() and ::operator new[]() can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program.

Class-specific memory allocation operators can also be overloaded. The operator **new** can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator **new** must return a **void*** and must have a size_t as its first argument. To overload the **new** operators, use the following prototypes declared in the new.h header file.

- void * operator new(size_t Type_size);   // For Non-array
- void * operator new[](size_t Type_size);   // For arrays

The C++Builder compiler provides *Type_size* to the **new** operator. Any data type may be substitued for *Type_size* except function names (although a pointer to function is permitted), class declarations, enumeration declarations, const, volatile.

Overloading the Operator delete

Example

The global operators, ::operator delete(), and **::operator delete[]()** cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the each global delete function can exist in the program.

The user-defined operator delete must have a **void** return type and **void*** as its first argument; a second argument of type *size_t* is optional. A class *T* can define at most one version of each of *T*::**operator delete[]()** and *T*::**operator delete()**. To overload the **delete** operators, use the following prototypes.

```
▪      void operator delete(void *Type_ptr, [size_t Type_size]);     // For
Non-array
▪      void operator delete[](size_t Type_ptr, [size_t Type_size]);  // For
arrays
```

Example of overloading new and delete

```
#include <stdlib.h>

class X {
   .
   .
   .
public:
   void* operator new(size_t size) { return newalloc(size);}
   void operator delete(void* p) { newfree(p); }
   X() { /* initialize here */ }
   X(char ch) { /* and here */ }

   ~X() { /* clean up here */ }
   .
   .
   .
};
```

Note: Destructors are called only if you use the **-xd** compiler option and an exception is thrown.

The *size* argument gives the size of the object being created, and *newalloc* and *newfree* are user-supplied memory allocation and deallocation functions. Constructor and destructor calls for objects of class *X* (or objects of classes derived from *X* that do not have their own overloaded operators **new** and **delete**) will invoke the matching user-defined *X*::**operator new()** and *X*::**operator delete()**, respectively. (Destructors will be called only if you use the **-xd** compiler option and an exception is thrown.)

The *X*::**operator new()**, *X*::**operator new[]()**, *X*::**operator delete()** and *X*::**operator delete[]()** operator functions are static members of *X* whether explicitly declared as static or not, so they cannot be virtual functions.

The standard, predefined (global) **::operator new()**, **::operator new[]()**, **:: operator delete()**, and **::operator delete[]()** operators can still be used within the scope of *X*, either explicitly with the global scope or implicitly when creating and destroying non-*X* or non-*X*-derived class objects. For example, you could use the standard **new** and **delete** when defining the overloaded versions:

```
void* X::operator new(size_t s)
{
   void* ptr = new char[s]; // standard new called
     .
     .
     .
   return ptr;
}
```

```
void X::operator delete(void* ptr)
{
     .
     .
     .
   delete (void*) ptr;     // standard delete called
}
```

The reason for the *size* argument is that classes derived from *X* inherit the *X*::**operator new()** and *X*::**operator new[]()**. The size of a derived class object may well differ from that of the base class.

Classes

See also

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that *inherit* the members of one or more *base* (or parent) classes.

In C++, structures and unions are considered as classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

class-key <*type-info*> *class-name*

<: *base-list*> { <*member-list*> };

class-key is one of **class**, **struct**, or **union**.

The optional *type-info* indicates a request for run-time type information about the class. You can compile with the **–RT** compiler option, or you can use the **_ _rtti** keyword. See the discussion of class typeinfo for more information.

The optional *base-list* lists the base class or classes from which the class *class-name* will derive (or *inherit*) objects and methods. If any base classes are specified, the class *class-name* is called a derived class. The *base-list* has default and optional overriding *access specifiers* that can modify the access rights of the derived class to members of the base classes.

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that can affect which functions can access which members.

VCL class declarations

See also Example Keywords

Syntax

```
__declspec(<decl-modifier>)
```

Description

The *decl-modifier* argument can be one of *delphireturn* , *delphiclass*, or *pascalimplementation*. These arguments should be used only with classes derived from VCL classes.

- You must use *__declspec(delphiclass)* for any forward declaration of classes that are directly or indirectly derived from *TObject*.
- You must use *__declspec(delphireturn)* when you make forward declarations of classes that directly or indirectly derive from *Currency*, *AnsiString*, *Variant*, *TDateTime*, or *Set*.
- Use the *__declspec(pascalimplementation)* modifier to indicate that a class has been implemented in Object Pascal. This modifier appears in a Pascal portability header file with a .hpp extension.

The *delphireturn* argument is used to mark C++ classes for VCL-compatible handling in function calls as parameters and return values.

The *delphiclass* argument is used to create classes that have the following VCL compatibility.

- VCL-compatible RTTI
- VCL-compatible constructor/destructor behavior
- VCL-compatible exception handling

A VCL-compatible class has the following restrictions.

- No virtual base classes or multiple inheritance is allowed.
- Must be dynamically allocated by using the global **new** operator.
- Copy and assignment constructors must be explicitly defined. The compiler does not automatically provide these constructors for VCL-derived classes.

VCL class names

See also Keywords

The C++Builder IDE expects each VCL class to begin with the letter “T”. Therefore, the first T is always ignored when assigning the program id. For example, a `class Test` declaration, results in a program id without the leading `T`.

Class names

See also

class-name is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted. See Untagged structures and typedefs for discussion of untagged structures.

Class types

See also

The declaration creates a unique type, class type *class-name*. This lets you declare further *class objects* (or *instances*) of this type, and objects derived from this type (such as pointers to, references to, arrays of *class-name*, and so on):

```
class X { ... };
X x, &xr, *xptr, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of X */
struct Y { ... };
Y y, &yr, *yptr, yarray[10];
// C would have
// struct Y y, *yptr, yarray[10];
union Z { ... };
Z z, &zr, *zptr, zarray[10];
// C would have
// union Z z, *zptr, zarray[10];
```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++, they are needed only when the class names, Y and Z, are hidden (see Class name scope)

Class name scope

See also

The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the *elaborated type specifier*. This means that the class key, **class**, **struct**, or **union**, must be used with the class name. For example,

```
struct S { ... };
int S(struct S *Sptr);
void func(void) {
   S t;         // ILLEGAL declaration: no class key and function S in scope
   struct S s;  // OK: elaborated with class key
   S(&s);       // OK: this is a function call
}
```

C++ also allows a forward class declaration:

```
class X;  // no members, yet!
```

Forward declarations permit certain references to class name *X* (usually references to pointers to class objects) before the class has been fully defined. See Structure member declarations for more information. Of course, you must make a complete class declaration with members before you can define and use class objects.

Forward declarations cannot be made for typedef classes.

See also the syntax for forward declarations of VCL classes.

Class objects

See also

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class.

Redefined functions and operators are said to be *overloaded*. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called *member functions* for that class. C++ offers the overloading mechanism that allows the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

Class member list

See also

The optional *member-list* is a sequence of data declarations (of any type, including enumerations, bit fields and other classes), function declarations, and definitions, all with optional storage class specifiers and access modifiers. The objects thus defined are called *class members*. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

Member functions

See also Example

A function declared without the **friend** specifier is known as a *member function* of the class. Functions declared with the **friend** modifier are called *friend functions*.

Member functions are often referred to as *methods* in Object Pascal and Delphi documentation.

The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.

Member function access

```
/* Getting access to member functions of a base class. */
#include <iostream.h>
class X {
public:
   void func1() {cout << "func1" << endl; }
};

class Y : public X {
public:
   void func2()
   {
    cout << "func2" << endl;
    }
};

void main() {
 X myX;
 Y *myY = new Y;
   myX.func1();
   myY ->func2();   // Member function
   myY ->func1();   // Member function in base class
}
```

The keyword this

See also

Nonstatic member functions operate on the class type object they are called with. For example, if *x* is an object of class *X* and *f()* is a member function of *X*, the function call `x.f()` operates on *x*. Similarly, if *xptr* is a pointer to an *X* object, the function call `xptr->f()` operates on *xptr. But how does *f* know which instance of *X* it is operating on? C++ provides *f* with a pointer to *x* called **this**. **this** is passed as a hidden argument in all calls to nonstatic member functions.

this is a local variable available in the body of any nonstatic member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If *x.f(y)* is called, for example, where *y* is a member of *X*, **this** is set to *&x* and *y* is set to **this->***y*, which is equivalent to *x.y*.

Static members

See also

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called *static members* and have distinct properties from nonstatic members. With nonstatic members, a distinct copy "exists" for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If *x* is a static member of class *X*, it can be referenced as *X::x* (even if objects of class *X* haven't been created yet). It is still possible to access x using the normal member access operators. For example, *y.x* and *yptr->x*, where *y* is an object of class *X* and *yptr* is a pointer to an object of class *X*, although the expressions *y* and *yptr* are not evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
   int member_int;
public:
   static void func(int i, X* ptr);
};
void g(void); {
   X obj;
   func(1, &obj);       // error unless there is a global func()
                        // defined elsewhere
   X::func(1, &obj);    // calls the static func() in X
                        // OK for static functions only
   obj.func(1, &obj);   // so does this (OK for static and
                        // nonstatic functions)
}
```

Because static member functions can be called with no particular object in mind, they don't have a **this** pointer, and therefore cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, *func* might be defined as follows:

```
void X::func(int i, X* ptr)
{
   member_int = i;        // which object does member_int
                          // refer to? Error
   ptr->member_int = i;   // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members, nested to any level, obey the usual class member access rules, except they can be initialized.

```
class X {
   static int x;
   static const int size =  5;
   class inner {
      static float f;
      void func(void);     // nested declaration
      };
public :
   char array[size];
};
```

```
int X::x = 1;
float X::inner::f = 3.14;  // initialization of nested static
X::inner::func(void) {     /*  define the nested function */  }
```

The principal use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- Reduce the number of visible global names
- Make obvious which static objects logically belong to which class
- Permit access control to their names

Inline functions

See also

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an *inline function*.

C++Builder can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an *inline expansion* of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The **inline** specifier indicates to the compiler you would like an inline expansion.

Note: The C++Builder compiler can ignore requests for inline expansion.

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the operator functions that implement overloaded operators. For example, the following class declaration of *func*:

```
int i;                          // global int
class X {
public:
   char* func(void) { return i; }  // inline by default
   char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined outside the class with an explicit **inline** specifier. The *i* returned by *func* is the **char*** *i* of class *X* (see Member scope).

Inline functions and exceptions

An inline function with an exception-specification will never be expanded inline by C++Builder. For example,

```
inline void f1() throw(int)
   {
   // Warning: Functions with exception specifications are not expanded inli
  ne
   }
```

The remaining restrictions apply only when destructor cleanup is enabled.

Note: Destructors are called by default. See Setting Exception Handling Options for information about exception-handling switches.

An inline function that takes at least one parameter that is of type 'class with a destructor' will not be expanded inline. Note that this restriction does not apply to classes that are passed by reference. Example:

```
struct foo {
   foo();
   ~foo();
   };
inline void f2(foo& x) {
   // no warning, f2() can be expanded inline
   }
inline void f3(foo x) {
   // Warning: Functions taking class-by-value argument(s) are
   //          not expanded inline in function f3(foo)
   }
```

An inline function that returns a class with a destructor by value will not be expanded inline whenever there are variables or temporaries that need to be destructed within the return expression:

```
struct foo {
   foo();
   ~foo();
   };
inline foo f4() {
   return foo();
   // no warning, f4() can be expanded inline
   }
inline foo f5() {
   foo X;
   return foo(); // Object X needs to be destructed
   // Warning: Functions containing some return statements are
   //          not expanded inline in function f5()
   }
inline foo f6() {
   return ( foo(), foo() );  // temporary in return value
   // Warning:   Functions containing some return statements are
   //            not expanded inline in function f6()
   }
```

Member scope

See also

The expression `X::func()` in the example in Inline functions and exceptions uses the class name *X* with the scope access modifier to signify that *func*, although defined “outside” the class, is indeed a member function of *X* and exists within the scope of *X*. The influence of *X*:: extends into the body of the definition. This explains why the *i* returned by *func* refers to *X::i*, the **char*** *i* of *X*, rather than the global **int** *i*. Without the *X*:: modifier, the function func would represent an ordinary non-class function, returning the global **int** *i*.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class *X* can be referenced using the selection operators **.** and **->** (as with C structures). Member functions can also be called using the selection operators (see The keyword this). For example:

```
class X {
public:
   int i;
   char name[20];
   X *ptr1;
   X *ptr2;
   void Xfunc(char*data, X* left, X* right);   // define elsewhere
};
void f(void);
{
   X x1, x2, *xptr=&x1;
   x1.i = 0;
   x2.i = x1.i;
   xptr->i = 1;
   x1.Xfunc("stan", &x2, xptr);
}
```

If *m* is a member or base member of class *X*, the expression *X::m* is called a *qualified name*; it has the same type as *m*, and it is an lvalue only if *m* is an lvalue. It is important to note that, even if the class name *X* is hidden by a non-type name, the qualified name *X::m* will access the correct class member, *m*.

Class members cannot be added to a class by another section of your program. The class *X* cannot contain objects of class *X*, but can contain pointers or references to objects of class *X* (note the similarity with C’s structure and union types).

Nested types

See also

Tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the ***xxx::yyy*** notation, except when in the scope of the appropriate class.

A class declared within another class is called a *nested class*. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).

Classes can be nested in this way to an arbitrary level. Nested classes can be declared inside some class and defined later. For example,

```
struct outer
{
   typedef int t;  // 'outer::t' is a typedef name
   struct inner    // 'outer::inner' is a class
   {
      static int x;
   };
   static int x;
      int f();
   class deep;     // nested declaration
};
int outer::x;      // define static data member
int outer::f() {
   t x;            // 't' visible directly here
   return x;
   }
int outer::inner::x;    // define static data member
outer::t x;             //  have to use 'outer::t' here
class outer::deep { };  // define the nested class here
```

With Borland C++ 2.0, any tags or **typedef** names declared inside a class actually belong to the global (file) scope. For example:

```
struct foo
{
   enum bar { x };   // 2.0 rules: 'bar' belongs to file scope
                     // 2.1 rules: 'bar' belongs to 'foo' scope
};
bar x;
```

The preceding fragment compiles without errors. But because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Member access control

See also

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

- **public**: The member can be used by any function.
- **private**: The member can be used only by member functions and friends of the class it's declared in.
- **protected**: Same as for **private**. Additionally, the member can be used by member functions and friends of classes *derived* from the declared class, but only in objects of the derived type. (Derived classes are explained in Base and derived class access.)

Note: Friend function declarations are not affected by access specifiers (see Friends of classes for more information).

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
   int i;    // X::i is private by default
   char ch;  // so is X::ch
public:
   int j;    // next two are public
   int k;
protected:
   int l;    // X::l is protected
};
struct Y {
   int i;    // Y::i is public by default
private:
   int j;    // Y::j is private
public:
   int k;    // Y::k is public
};
union Z {
   int i;    // public by default; no other choice
   double d;
};
```

Note: The access specifiers can be listed and grouped in any convenient sequence. You can save typing effort by declaring all the private members together, and so on.

Base and derived class access

See also

When you declare a derived class *D*, you list the base classes *B1, B2*, ... in a comma-delimited *base-list*:

```
class-key D : base-list { <member-list> }
```

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) *D* can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by *D*? *D* might want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the *base-list*.

Note: Since a base class can itself be a derived class, the access attribute question is recursive: you backtrack until you reach the basest of the base classes, those that do not inherit.

When declaring *D*, you can use the access specifier **public**, **protected**, or **private** in front of the classes in the *base-list*:

```
class D : public B1, private B2, ... {
   .
   .
   .
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they can alter the access attributes of base members as viewed by the derived class.

The default is private if D is a class declaration, and public if D is a struct declaration.

Note: Unions cannot have base classes, and unions cannot be used as base classes.

The derived class inherits access attributes from a base class as follows:

- **public** base class: **public** members of the base class are **public** members of the derived class. **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **protected** base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **private** base class: Both **public** and **protected** members of the base class are **private** members of the derived class. **private** members of the base class remain **private** to the base class.

Note that **private** members of a base class are always inaccessible to member functions of the derived class *unless* **friend** declarations are explicitly declared in the base class granting access. For example,

```
/* class X is derived from class A */
class X : A {               // default for class is private A
   .
   .
   .
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */
class Y : B, public C {     // override default for C
   .
   .
   .
}
/* struct S is derived from D */
struct S : D {              // default for struct is public D
   .
   .
   .
}
```

```
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
struct T : private D, E {    // override default for D
                             // E is public by default
  .
  .
  .
}
```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations of the derived class. For example:

```
class B {
   int a;               // private by default
public:
   int b, c;
   int Bfunc(void);
};
class X : private B {   // a, b, c, Bfunc are now private in X
   int d;               // private by default, NOTE: a is not
                        // accessible in X
public:
   B::c;                // c was private, now is public
   int e;
   int Xfunc(void);
};
int Efunc(X& x);        // external to B and X
```

The function *Efunc()* can use only the public names *c*, *e*, and *Xfunc()*.

The function *Xfunc()* is in *X*, which is derived from **private** *B*, so it has access to

- The “adjusted-to-public” *c*
- The “private-to-*X*” members from *B*: *b* and *Bfunc()*
- X’s own private and public members: *d*, *e*, and *Xfunc()*

However, *Xfunc()* cannot access the “private-to-*B*” member, *a*.

Virtual base classes

See also

A **virtual** class is a base class that is passed to more than one derived class, as might happen with multiple inheritance.

You cannot specify a base class more than once in a derived class:

```
class B { ...};
class D : B, B { ... };  // ILLEGAL
```

However, you can indirectly pass a base class to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }  // OK
```

In this case, each object of class Z has two sub-objects of class B.

If this causes problems, add the keyword **virtual** to the base class specifier. For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes.

If the hierarchy contains multiple virtual base classes, the virtual base class constructors invoke in the order they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly constructed. For example, this code

```
class X : public Y, virtual public Z
   X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // non-virtual base class
X();   // derived class
```

Friends of classes

See also

A **friend** *F* of a class *X* is a function or class, although not a member function of *X*, with full access rights to the private and protected members of *X*. In all other respects, *F* is a normal function with respect to scope, declarations, and definitions.

Since *F* is not a member of *X*, it is not in the scope of *X*, and it cannot be called with the *x.F* and *xptr->F* selector operators (where *x* is an *X* object and *xptr* is a pointer to an *X* object).

If the specifier **friend** is used with a function declaration or definition within the class *X*, it becomes a friend of *X*.

friend functions defined within a class obey the same inline rules as member functions (see Inline functions). **friend** functions are not affected by their position within the class or by any access specifiers. For example:

```
class X {
   int i;                           // private to X
   friend void friend_func(X*, int);
/* friend_func is not private, even though it's declared in the private sect
  ion */
public:
     void member_func(int);
};
/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;
/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class Y into friends of class X with a single declaration:

```
class Y;                      // incomplete declaration
class X {
   friend Y;
   int i;
   void member_funcX();
};
class Y; {                    // complete the declaration
   void friend_X1(X&);
   void friend_X2(X*);
     .
     .
     .
};
```

The functions declared in *Y* are friends of *X*, although they have no **friend** specifiers. They can access the private members of *X*, such as *i* and *member_funcX*.

It is also possible for an individual member function of class *X* to be a friend of class *Y*:

```
class X {
     .
     .
     .
   void member_funcX();
}
class Y {
   int i;
   friend void X::member_funcX();
     .
     .
     .
};
```

Class friendship is not transitive: *X* friend of *Y* and *Y* friend of *Z* does not imply *X* friend of *Z*. Friendship is not inherited.

Introduction to constructors and destructors

See also

There are several special member functions that determine how the objects of a class are created, initalized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features:

- They do not have return value declarations (not even **void**).
- They cannot be inherited, though a derived class can call the base class's constructors and destructors.
- Constructors, like most C++ functions, can have default arguments or use member initialization lists.
- Destructors can be **virtual**, but constructors cannot. (See Virtual destructors.)
- You can't take their addresses.

```
int main (void)
{
      .
      .
      .
      void *ptr = base::base;     // illegal
      .
      .
      .
}
```

- Constructors and destructors can be generated by C++Builder if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.
- You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
      .
      .
      .
      X *p;
      .
      .
      .
      p->X::~X();                 // legal call of destructor
      X::X();                     // illegal call of constructor
      .
      .
      .
}
```

- The compiler automatically calls constructors and destructors when defining and destroying objects.
- Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
- An object with a constructor or destructor cannot be used as a member of a union.
- If no constructor has been defined for some class *X* to accept a given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class *X*. Note that this rule applies only to any constructor with *one* parameter and no initializers that use the "**=**" syntax.

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;                    // illegal: Y(X(1)) not tried
```

If **class** *X* has one or more constructors, one of them is invoked each time you define an object *x* of **class** *X*. The constructor creates x and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

Constructors

See also

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the *main* function is called. When the **#pragma startup** directive is used to install a function prior to the *main* function, global variable constructors are called prior to the startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {
public:
   X();   // class X constructor
};
```

A **class** *X* constructor cannot take *X* as an argument:

```
class X {
public:
   X(X);                        // illegal
};
```

The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the copy constructor . A constructor that accepts no parameters is called the default constructor .

Constructor defaults

See also

The default constructor for **class** *X* is one that takes no arguments; it usually has the form `X::X()`. If no user-defined constructors exist for a class, C++Builder generates a default constructor. On a declaration such as *X x*, the default constructor creates the object *x*.

Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes no arguments and must not be confused with, say, `X::X(int = 0)`, which can be called with *no* arguments as a default constructor, or can take an argument.

You should avoid ambiguity in defining constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
   X();
   X(int i = 0);
};
int main() {
   X one(10);  // OK; uses X::X(int)
   X two;      // Error;ambiguous whether to call X::X() or
               // X::X(int = 0)
   return 0;
}
```

The copy constructor

See also

A copy constructor for **class** *X* is one that can be called with a single argument of type *X* as follows:

```
X::X(X&)
```

or

```
X::X(const X&)
```

or

```
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x1;
X x2 = x1;
X x3(x1);
```

C++Builder generates a copy constructor for **class** *X* if one is needed and no other constructor has been defined in **class** *X*. The copy constructor that is generated by the C++Builder compiler lets you safely start programming with simple data types. You need to make your own definition of the copy constructor if your program creates aggregate, complex types such as **class**, **struct**, and array types. The copy constructor is also called when you pass a class argument by value to a function.

See also the discussion of member-by-member class assignment. You should define the copy constructor if you overload the assignment operator.

Overloading constructors

See also

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X {
   int    integer_part;
   double double_part;
public:
   X(int i)    { integer_part = i; }
   X(double d) { double_part = d; }
};
int main() {
   X one(10);   // invokes X::X(int) and sets integer_part to 10
   X one(3.14); // invokes X::X(double) setting double_part to 3.14
   return 0;
}
```

Order of calling constructors

See also

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y();   // base class constructor
X();   // derived class constructor
```

For the case of multiple base classes,

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y();  // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any nonvirtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any nonvirtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a nonvirtual base, that nonvirtual base will be first so that the virtual base class can be properly constructed. The code:

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // nonvirtual base class
X();   // derived class
```

Or, for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base();       // virtual base class highest in hierarchy
              // base is constructed only once
base2();      // nonvirtual base of virtual base level2
              // must be called to construct level2
level2();     // virtual base class
base2();      // nonvirtual base of level1
level1();     // other nonvirtual base
toplevel();
```

If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

Class initialization

See also

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
   int i;
public:
   X();          // function bodies omitted for clarity
   X(int x);
   X(const X&);
};
void main()
{
   X one;         // default constructor invoked
   X two(1);      // constructor X::X(int) is used
   X three = 1;   // calls X::X(int)
   X four = one;  // invokes X::X(const X&) for copy
   X five(two);   // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways:

- It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```
class X
{
   int a, b;
public:
   X(int i, int j) { a = i; b = j }
};
```

- An initializer list can be used prior to the function body:

```
class X
{
   int a, b, &c;  // Note the reference variable.
public:
   X(int i, int j) : a(i), b(j), c(a) {}
};
```

The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to *x::a* and 2 to *x::b*. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

Note: Base class constructors must be declared as either **public** or **protected** to be called from a derived class.

```
class base1
{
   int x;
public:
   base1(int i) { x = i; }
};

class base2
{
   int x;
public:
   base2(int i) : x(i) {}
};
class top : public base1, public base2
{
   int a, b;
public:
   top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j;}
};
```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of *base1* with the value 5 and *base2* with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```
class X
{
  int a, b;
public:
  X(int i, j) :  a(i), b(a+j) {}
};
```

With this class, a declaration of `X x(1,1)` results in an assignment of 1 to *x::a* and 2 to *x::b*.

Base class constructors are called prior to the construction of any of the derived classes members. If the values of the derived class are changed, they will have no effect on the creation of the base class.

```
class base
{
   int x;
public:
   base(int i) : x(i) {}
};
class derived : base
{
   int a;
public:
   derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                         // passed an uninitialized 'a'
};
```

With this class setup, a call of `derived d(1)` will *not* result in a value of 10 for the base class member *x*. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
   .
   .
   .
}
```

Destructors

See also

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
   ~X(){};  // destructor for class X
};
```

If a destructor isn't explicitly defined for a class, the compiler generates one.

Invoking destructors

See also

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see Order of calling constructors).

atexit, #pragma exit, and destructors

See also

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the *main* function, are destroyed as they go out of scope. The order of execution at the end of a C++Builder program is as follows:

- *atexit()* functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

exit and destructors

See also

When you call *exit* from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

abort and destructors

See also

If you call *abort* anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to **new**

```
#include <stdlib.h>
class X {
public:
  .
  .
  .
   ~X(){};
  .
  .
  .
};
void* operator new(size_t size, void *ptr)
{
   return ptr;
}
char buffer[sizeof(X)];
void main() {
   X* pointer = new X;
   X* exact_pointer;
   exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer
     .
     .
     .
   delete pointer;                 // delete used to destroy pointer
   exact_pointer->X::~X();         // direct call used to deallocate
}
```

Virtual destructors

See also

A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

```
/* How virtual affects the order of destructor calls.
   Without a virtual destructor in the base class, the derived
   class destructor won't be called. */
#include <iostream.h>
class color {
public:
   virtual ~color() {  // Virtual destructor
      cout << "color dtor\n";
      }
};
class red : public color {
public:
   ~red() {  // This destructor is also virtual
      cout << "red dtor\n";
      }
};
class brightred : public red {
public:
   ~brightred() {  // This destructor is also virtual
       cout << "brightred dtor\n";
       }
};
int main() {
   color *palette[3];
   palette[0] = new red;
   palette[1] = new brightred;
   palette[2] = new color;

   // The destructors for red and color are called.
   delete palette[0];
   cout << endl;

   // The destructors for bright red, red, and color are called.
   delete palette[1];
   cout << endl;

   // The destructor for color is called.
   delete palette[2];
   return 0;
}
```

Program Output:

```
red dtor
color dtor

brightred dtor
red dtor
color dtor

color dtor
```

However, if no destructors are declared as virtual, **delete** *palette*[0], **delete** *palette*[1], and **delete** *palette*[2] would all call only the destructor for class *color*. This would incorrectly destruct the first two elements, which were actually of type *red* and *brightred*.

Overloading Operators

See also Operators

C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators can be overloaded except for:

```
.     .*    ::    ?:
```

The followoing preprocessing symbols cannot be overloaded.

```
  #     ##
```

The =, [ ], ( ), and -> operators can be overloaded only as nonstatic member functions. These operators cannot be overloaded for **enum** types. Any attempt to overload a global version of these operators results in a compile-time error.

The keyword **operator** followed by the operator symbol is called the operator function name; it is used like a normal function name when defining the new (overloaded) action for the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function cannot alter the number of arguments or the precedence and associativity rules applying to normal operator use.

Example for Overloading Operators

The following example extends the class complex to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in the usual mathematical syntax.

Some of the issues illustrated by the example are:

- The default constructor is defined. This is provided by the compiler only if you have not defined it or any other constructor.
- The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator.
- The assignment operator is overloaded. If you do not overload the assignment operator, the compiler calls a default assignment operator when required. By overloading assignment of *cvector* types, you specify exactly the actions to be taken. Note that the assignment operator function cannot be inherited by derived classes
- The subscript operator is defined as a member function (a requirement when overloading) with a single argument. The **const** version assures the caller that it will not modify its argument—this is useful when copying or assigning. This operator should check that the index value is within range—a good place to implement exception handling.
- The addition operator is defined as a member function. It allows addition only for *cvector* types. Addition should always check that the operands' sizes are compatible.
- The multiplication operator is declared a **friend**. This lets you define the order of the operands. An attempt to reverse the order of the operands is a compile-time error.
- The stream insertion operator is overloaded to naturally display a *cvector*. Large objects that don't display well on a limited size screen might require a different display strategy.

Source

```
/* HOW TO EXTEND THE complex CLASS AND OVERLOAD THE REQUIRED OPERATORS. */
#pragma warn -inl     // IGNORE not expanded inline WARNINGS.
#include <complex.h>  // THIS ALREADY INCLUDES iostream.h
// COMPLEX VECTORS
class cvector {
   int size;
   complex *data;
public:
   cvector() { size = 0; data = NULL; };
   cvector(int i = 5) : size(i) {    // DEFAULT VECTOR SIZE.
      data = new complex[size];
      for (int j = 0; j < size; j++)
           data[j] = j + (0.1 * j);  // ARBITRARY INITIALIZATION.
      };
   /* THIS VERSION IS CALLED IN main() */
   complex& operator [](int i) { return data[i]; };
   /* THIS VERSION IS CALLED IN ASSIGNMENT OPERATOR AND COPY THE CONSTRUCTOR
  */
   const complex& operator [](int i) const { return data[i]; };
   cvector operator +(cvector& A) {  // ADDITION OPERATOR
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < size; i++)
          result[i] = data[i] + A.data[i];
      return result;
      };
   /* BECAUSE scalar * vector MULTIPLICATION IS NOT COMMUTATIVE, THE ORDER O
  F
      THE ELEMENTS MUST BE SPECIFIED. THIS FRIEND OPERATOR FUNCTION WILL ENS
  URE
      PROPER MULTIPLICATION. */
   friend cvector operator *(int scalar, cvector& A) {
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < A.size; i++)
        result.data[i] = scalar * A.data[i];
      return result;
      }
   /* THE STREAM INSERTION OPERATOR. */
   friend ostream& operator <<(ostream& out_data, cvector& C) {
      for (int i = 0; i < C.size; i++)
           out_data << "[" << i << "]=" << C.data[i] << "   ";
      cout << endl;
      return out_data;
      };
   cvector( const cvector &C ) {  // COPY CONSTRUCTOR
      size = C.size;
      data = new complex[size];
      for (int i = 0; i < size; i++)
           data[i] = C[i];
     }
   cvector& operator =(const cvector &C) { // ASSIGNMENT OPERATOR.
      if (this == &C) return *this;
      delete[] data;
      size = C.size;
      data = new complex[size];
      for (int i = 0; i < size; i++)
          data[i] = C[i];
      return *this;
   };
   virtual ~cvector() { delete[] data; }; // DESTRUCTOR
   };
int main(void) { /* A FEW OPERATIONS WITH complex VECTORS. */
   cvector cvector1(4), cvector2(4), result(4);
   // CREATE complex NUMBERS AND ASSIGN THEM TO complex VECTORS
   cvector1[3] = complex(3.3, 102.8);
   cout << "Here is cvector1:" << endl;
   cout << cvector1;
   cvector2[3] = complex(33.3, 81);
   cout << "Here is cvector2:" << endl;
   cout << cvector2;
   result = cvector1 + cvector2;
   cout << "The result of vector addition:" << endl;
   cout << result;
   result = 10 * cvector2;
   cout << "The result of 10 * cvector2:" << endl;
   cout << result;
   return 0;
   }
```

Output

```
Here is cvector1:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(3.3, 102.8)
Here is cvector2:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(33.3, 81)
The result of vector addition:
[0]=(0, 0)   [1]=(2.2, 0)   [2]=(4.4, 0)   [3]=(36.6, 183.8)
```

```
The result of 10 * cvector2:
[0]=(0, 0)   [1]=(11, 0)   [2]=(22, 0)   [3]=(333, 810)
```

Overloading Operator Functions

See also Operators

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2);   // same as c3 = c1 + c2
```

Apart from new and delete, which have their own rules, an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions =, ( ), [ ] and -> must be nonstatic member functions.

Enumerations can have overloaded operators. However, the operator functions =, ( ), [ ], and -> cannot be overloaded for enumerations.

Overloaded Operators and Inheritance

See also Operators

With the exception of the assignment function operator =( ), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could possibly be further overloaded for Y.

Overloading Unary Operators

See also Operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either x.operator@() or operator@(x), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

- Under C++ 2.0, an overloaded operator ++ or -- is used for both prefix and postfix uses of the operator.
- With C++ 2.1, when an operator++ or operator- - is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix operator++ or operator- -. You can only overload a postfix operator++ or operator- - by defining it as a member function taking an int parameter or as a nonmember function taking one class and one int parameter.

When only the prefix version of an operator++ or operator- - is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator in
  function func()
Warning: Overloaded prefix 'operator --' used as a postfix operator in
  function func()
```

Overloading Binary Operators

See also Operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually friend) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either x.operator@(y) or operator@(x,y) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Overloading the Assignment Operator =

See also Operators

The assignment operator=( ) can be overloaded by declaring a nonstatic member function. For example,

```
class String {
     .
     .
     .
   String& operator = (String& str);
     .
     .
     .
   String (String&);
   ~String();
}
```

This code, with suitable definitions of String::operator =(), allows string assignments str1 = str2 in the usual sense. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class X, there is no user-defined operator =, the operator = is defined by default as a member-by-member assignment of the members of class X:

```
X& X::operator = (const X& source)
{
   // memberwise assignment
}
```

Overloading the Function Call Operator ( )

See also Operators

Syntax

```
postfix-expression ( <expression-list> )
```

Description

In its ordinary use as a function call, the postfix-expression must be a function name, or a pointer or reference to a function. When the postfix-expression is used to make a member function call, postfix-expression must be a class member function name or a pointer-to-member expression used to select a class member function. In either case, the postfix-expression is followed by the optional expression-list (possibly empty).

A call *X(arg1, arg2)*, where *X* is an object class *X*, is interpreted as *X*.**operator()**(*arg1, arg2*).

The function call operator, **operator()()**, can only be overloaded as a nonstatic member function.

Overloading the Subscript Operator [ ]

Syntax

```
postfix-expression [ expression ]
```

Description

The corresponding operator function is operator[]() this can be user-defined for a class *X* (and any derived classes). The expression *X*[*y*], where *X* is an object of class *X*, is interpreted as `x.operator[](y)`.

The **operator[]()** can only be overloaded as a nonstatic member function.

Overloading the Class Member Access Operator ->

See also Operators

Syntax

```
postfix-expression -> primary-expression
```

Description

The expression `x->m`, where *x* is a **class** *X* object, is interpreted as `(x.operator->())->m`, so that the function **operator->()** must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

The **operator->()** can only be overloaded as a nonstatic member function.

Polymorphic classes

See also

Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are **class** and **struct**.

Virtual functions

See also

virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the **virtual** keyword. To declare a *pure* function (which automatically declares an abstract class), prefix the prototype with the **virtual** keyword, and set the function equal to zero.

```
virtual int funct1(void);       // A virtual function declaration.
virtual int funct2(void) = 0;   // A pure function declaration.
virtual void funct3(void) = 0 { // This is a valid declaration.
   // Some code here.
   };
```

Note: See Abstract classes for a discussion of pure virtual functions.

When you declare **virtual** functions, keep these guidelines in mind:

- They can be member functions only.
- They can be declared a **friend** of another class.
- They cannot be a static member.

A **virtual** function does not need to be redefined in a derived class. You can supply one definition in the base class so that all calls will access the base function.

To redefine a **virtual** function in any derived class, the number and type of arguments must be the same in the base class declaration and in the derived class declaration. (The case for redefined **virtual** functions differing only in return type is discussed below.) A redefined function is said to *override* the base class function.

You can also declare the functions `int Base::Fun(int)` and `int Derived::Fun(int)` even when they are not virtual. In such a case, `int Derived::Fun(int)` is said to hide any other versions of `Fun(int)` that exist in any base classes. In addition, if class *Derived* defines other versions of *Fun()*, (that is, versions of *Fun()* with different signatures) such versions are said to be overloaded versions of *Fun()*.

Virtual function return types

Generally, when redefining a **virtual** function, you cannot change just the function return type. To redefine a **virtual** function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the **virtual** function mechanism is ignored.

However, for certain virtual functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when *both* of the following conditions are met:

- The overridden **virtual** function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class *B* and class *D* (derived publicly from *B*) each contain a **virtual** function *vf*, then if *vf* is called for an object *d* of *D*, the call made is `D::vf()`, even when the access is via a pointer or reference to *B*. For example,

```
struct X {};      // Base class.
struct Y : X {};       // Derived class.
struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
   virtual X* pf();    // Return type is a pointer to base. This can
    //  be overridden.
   };
```

```
class D : public B {
public:
   virtual void vf1(); // Virtual specifier is legal but redundant.
   void vf2(int);      // Not virtual, since it's using a different
     //  arg list. This hides B::vf2().
// char vf3();         // Illegal: return-type-only change!
   void f();
   Y*   pf();          // Overriding function differs only
     //  in return type. Returns a pointer to
     //  the derived class.
   };
void extf() {
   D d;  // Instantiate D
   B* bp = &d; // Standard conversion from D* to B*
     // Initialize bp with the table of functions
       // provided for object d. If there is no entry for a
     // function in the d-table, use the function
       //  in the B-table.
   bp->vf1();  // Calls D::vf1
   bp->vf2();  // Calls B::vf2 since D's vf2 has different args
   bp->f();    // Calls B::f (not virtual)
   X* xptr = bp->pf();   // Calls D::pf() and converts the result
     //  to a pointer to X.
   D* dptr = &d;
   Y* yptr = dptr->pf(); // Calls D::pf() and initializes yptr.
     //  No further conversion is done.
   }
```

The overriding function *vf1* in *D* is automatically **virtual**. The **virtual** specifier *can* be used with an overriding function declaration in the derived class. If other classes will be derived from *D*, the **virtual** keyword is required. If no further classes will be derived from *D*, the use of **virtual** is redundant.

The interpretation of a **virtual** function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding).

Abstract classes

See also

An abstract class is a class with at least one pure **virtual** function. A **virtual** function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {       // abstract class
   point center;
     .
     .
     .
public:
   where() { return center; }
   move(point p) { center = p; draw(); }
   virtual void rotate(int) = 0; // pure virtual function
   virtual void draw() = 0;      // pure virtual function
   virtual void hilite() = 0;    // pure virtual function
     .
     .
     .
}
shape x; // ERROR: attempt to create an object of an abstract class
   shape* sptr;   // pointer to abstract class is OK
   shape f();     // ERROR: abstract class cannot be a return type
int g(shape s);   // ERROR: abstract class cannot be a function argument typ
  e
shape& h(shape&); // reference to abstract class as return
    // value or function argument is OK
```

Suppose that *D* is a derived class with the abstract class *B* as its immediate base class. Then for each pure virtual function pvf in *B*, if *D* doesn't provide a definition for *pvf*, *pvf* becomes a pure member function of *D*, and *D* will also be an abstract class.

For example, using the class *shape* previously outlined,

```
class circle : public shape {// circle derived from abstract class
   int radius;               // private
public:
   void rotate(int) { }      // virtual function defined: no action
                             //  to rotate a circle
   void draw();              // circle::draw must be defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a run-time error.

C++ scope

See also

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

Class scope

See also

The name *M* of a member of a class *X* has class scope “local to *X*”; it can be used only in the following situations:

- In member functions of *X*
- In expressions such as `x.M`, where *x* is an object of *X*
- In expressions such as *xptr->M*, where *xptr* is a pointer to an object of *X*
- In expressions such as `X::M` or `D::M`, where *D* is a derived class of *X*
- In forward references within the class of which it is a member

Names of functions declared as friends of *X* are not members of *X*; their names simply have enclosing scope.

Hiding

See also

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: `X::M`. A hidden file scope (global) name can be referenced with the unary operator :: (for example, *::g*). A class name *X* can be hidden by the name of an object, function, or enumerator declared within the scope of *X*, regardless of the order in which the names are declared. However, the hidden class name *X* can still be accessed by prefixing *X* with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name x is immediately after its complete declaration but before its initializer, if one exists.

C++ scoping rules summary

See also

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

- The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
- If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
- If the name is used outside any function and class, or is prefixed by the unary scope access operator ::, *and* if the name is not qualified by the binary :: operator or the member selection operators **.** and **->**, then the name must be a global object, function, or enumerator.
- If the name *n* appears in any of the forms *X::n, x.n* (where *x* is an object of *X* or a reference to X), or *ptr->n* (where *ptr* is a pointer to *X*), then *n* is the name of a member of X or the member of a class from which *X* is derived.
- Any name that hasn't been discussed yet and that is used in a static member function must either be declared in the block it occurs in or in an enclosing block, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks and global declarations of *n*. Names in different scopes are not overloaded.
- Any name that hasn't been discussed yet and that is used in a nonstatic member function of class *X* must either be declared in the block it occurs in or in an enclosing block, be a member of class *X* or a base class of *X*, or be a global name. The declaration of a local name *n* hides declarations of n in enclosing blocks, members of the function's class, and global declarations of *n*. The declaration of a member name hides declarations of the same name in base classes.
- The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a nondefining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
- A constructor initializer (see *ctor-initializer* in the class declarator syntax in C++Builder declaration syntax,) is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

Using Templates

See also

Templates, also called *generics* or *parameterized* types, let you construct a family of related functions or classes. These topics introduce the basic concept of templates:

template

See also Keywords

Syntax

```
template-declaration:
   template < template-argument-list > declaration

template-argument-list:
   template-argument
   template-argument-list, template argument

template-argument:
   type-argument
   argument-declaration

type-argument:
   class identifier

template-class-name:
   template-name < template-arg-list >

template-arg-list:
   template-arg
   template-arg-list , template-arg

template-arg:
   expression
   type-name

< template-argument-list > declaration
```

Description

Use templates (also called generics or parameterized types) to construct a family of related functions or classes.

Template body parsing

See also

Earlier versions of the compiler didn't check the syntax of a template body unless the template was instantiated. A template body is now parsed immediately when seen like every other declaration.

```
template <class T> class X : T
{
  Int  j;  // Error: Type name expected in template X<T>
};
```

Let's assume that *Int* hasn't yet been defined. This means that *Int* must be a member of the template argument *T*. But it also might just be a typing error and should be **int** instead of *Int*. Because the compiler can't guess the right meaning it issues an error message.

If you want to access types defined by a template argument you should use a **typedef** to make your intention clear to the compiler:

```
template <class T> class X : T
{
  typedef  T::Int  Int;
  Int  j;
};
```

You cannot just write

```
   typedef   T::Int;
```

as in earlier versions of the compiler. Not giving the **typedef** name was acceptable, but this now causes an error message.

All other templates mentioned inside the template body are declared or defined at that point. Therefore, the following example is ill-formed and will not compile:

```
template <class T> class  X
{
  void f(NotYetDefindedTemplate<T> x);
};
```

All template definitions must end with a semicolon. Earlier versions of the compiler did not complain if the semicolon was missing.

Function Templates

See also Using Templates

Consider a function *max(x, y)* that returns the larger of its two arguments. *x* and *y* can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters *x* and *y* to be declared at compile time. Without using templates, many overloaded versions of *max* are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the **#define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of *max(x, y)* is to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be. By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

```
template <class T> T max(T x, T y){
   return (x > y) ? x : y;
   };
```

The data type is represented by the template argument **<class T>**. When used in an application, the compiler generates the appropriate code for the *max* function according to the data type actually used in the call:

```
int i;
Myclass a, b;

int j = max(i,0);        // arguments are integers
Myclass m = max(a,b);    // arguments are type Myclass
```

Any data type (not just a class) can be used for **<class T>**. The compiler takes care of calling the appropriate **operator>()**, so you can use *max* with arguments of any type for which **operator>()** is defined.

Overriding a Template Function

Using Templates

The previous example is called a *function template* (or *generic function*, if you like). A specific instantiation of a function template is called a *template function*. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>

char *max(char *x, char *y){
   return(strcmp(x,y) > 0) ? x : y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

Implicit and Explicit Template Functions

Using Templates

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b){
        return  (a > b) ? a : b;
};

void    f(int i, char c){
        max(i, i);              // calls max(int ,int )
        max(c, c);              // calls max(char,char)
        max(i, c);              // no match for max(int,char)
        max(c, i);              // no match for max(char,int)
}
```

This code results in the following error messages:

```
Could not find a match for 'max(int,char)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)
```

If the user explicitly declares a template function, this function, on the other hand, will participate fully in overload resolution. See the example of explicit template function.

When searching for an exact match for template function parameters, trivial conversions are considered to be exact matches. See the example on trivial conversions.

Template functions with derived class pointer or reference arguments are permitted to match their public base classes. See the example of base class referencing.

Class Templates

See also Using Templates Example

A class template (also called a *generic class* or *class generator*) lets you define a pattern for class definitions. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a type parameter to the class, the system will generate type-safe class definitions on the fly.

As with function templates, an explicit *template class* definition can be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol *Vector* must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

Template Arguments

Using Templates

Multiple arguments are allowed as part of the class template declaration. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Non-type template arguments such as *size* can have default values. The value supplied for a non-type template argument must be a constant expression:

```
const int N = 128;
int i = 256;

Buffer<int, 2*N> b1;// OK
Buffer<float, i> b2;// Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

Using Angle Brackets in Templates

Using Templates

Be careful when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the > between *x* and 100 would prematurely close the template argument list.

Using Type-safe Generic Lists in Templates

Using Templates

In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

```
class GList
{
 public:
   void insert( void * );
   void *peek();
     .
     .
     .
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public Glist {
 public:
   void insert( Foo *f ) { GList::insert( f ); }
   Foo *peek() { return (Foo *)GList::peek(); }
       .
       .
       .
};
```

This is type-safe. *insert* will only take arguments of type pointer-to-*Foo* or object-derived-from-*Foo*, so the underlying container will only hold pointers that in fact point to something of type *Foo*. This means that the cast in *FooList::peek()* is always safe, and you've created a true *FooList*. Now, to do the same thing for a *BarList*, a *BazList*, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and be type-safe, you can once again use templates. See the example for type-safe generic list class.

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no run-time overhead imposed by this type safety.

Type-safe generic list class definition

```
template <class T> class List : public GList
{
public:
   void insert( T *t ) { GList::insert( t ); }
   T *peek() { return (T *)GList::peek(); }
     .
     .
     .
};

  // Create a List object of Foo types and name it fList.
  List<Foo> fList;

// Create a List object of Bar types and name it bList.
  List<Bar> bList;

// Create a List object of Baz types and name it zList.
  List<Baz> zList;
```

Eliminating Pointers in Templates

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of **virtual** function calls required, since the compiler knows the actual types of the objects. This is beneficial if the **virtual** functions are small enough to be effectively inlined. It's difficult to inline **virtual** functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

```
template <class T> aBase {
    .
    .
    .
 private:
  T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf> {
    .
    .
    .
};
```

All the functions in *aBase* can call functions defined in *aFilebuf* directly, without having to go through a pointer. And if any of the functions in *aFilebuf* can be inlined, you'll get a speed improvement, because templates allow them to be inlined.

Template Compiler Switches

Using Templates

The -Jg family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler.

For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which instances will be user-defined so that they aren't generated from the template.

Using template switches

When using the **-Jg** family of switches, there are two basic approaches for generating template instances:

Approach 1
Include the function body (for a function template) or member function and static data member definitions (for a template class) in the header file that defines the particular template, and use the default setting of the template switch (**-Jg**). If some instances of the template are user-defined, the declarations (prototypes, for example) for them should be included in the same header but preceded by **#pragma option -Jgx**. See the example for template header files.

Approach 2
Compile all of the source files comprising the program with the **-Jgx** switch (causing external references to templates to be generated). In order to provide the definitions for all of the template instances, add a file (or files) to the program that includes the template bodies (including any user-defined instance definitions), and list all the template instances needed in the rest of the program to provide the necessary public symbol definitions. Compile the file (or files) with the **-Jgd** switch. See the example for separate file template compilation.

Template header file

```
// Declare a template function and define it's body.
/* When this header file is included in a C++ source file, the sort template
  can be used without worrying about how the various instances are generated
  (with the exception of sort for integer arrays which is a user-defined
  instance. Its definition must be provided by the user. */
template<class T> void sort (T* array, int size)
{
   // Body of template goes here.
}
// Sorting of integer elements done by user-define instance.
#pragma option -Jgx
extern void sort(int *array, int size);
// Restore the template switch to its original state.
#pragma option -Jg
```

Separate file template compilation

```
// In vector.h
template <class elem, int size> class vector
{
   elem * value;
public:
   vector();
   elem & operator [ ] (int index) {
      return value[index];
      }
};
// In main.cpp source file.
#include "vector.h"
/** Let the compiler know that the following template instances will be
  defined elsewhere. **/
#pragma option -Jgx
// Use two instances of the vector template class.
vector<int, 100> int_100;
vector<char, 10> char_10;
int main( )
{
   return int_100[ 0 ] + char_10[ 0 ];
}

// In template.cpp source file.
#include <string.h>
#include "vector.h"
// Define any template bodies.
template <class elem, int size> vector <elem, size> :: vector()
{
   value = new elem[size];
   memset(value, 0, size * sizeof(elem) );
}
// Generate the necessary instances.
#pragma option -Jgd
typedef vector<int, 100> fake_int_100;
typedef vector<char, 10> fake_char_10;
```

Exporting and importing templates

See also

The declaration of a template function or template class needs to be sufficiently flexible to allow it to be used in either a DLL or an EXE file. The same template declaration should be available as an import and/or export, or without a modifier. To be completely flexible, the header file template declarations should not use **_ _export** or **_ _import** modifiers. This allows the user to apply the appropriate modifier at the point of instantiation depending on how the instantiation is to be used.

The following steps demonstrate exporting and importing of templates. The source code is organized in three files. Using the header file, code is generated in the DLL. A DLL library is created and linked to an EXE file.

1. Exportable/Importable Template Declarations

   The header file contains all template class and template function declarations. An export/import version of the templates can be instantiated by defining the appropriate macro at compile time.
2. Compiling Exportable Templates

   Write the source code for a DLL. When compiled, this DLL has reusable export code for templates.
3. Using ImportTemplates

   Now you can write a calling function that uses templates. This file is linked to the DLL. Only objects that are not declared in the header file and which are instantiated in the *main* function cause the compiler to generate new code. Code for a newly instantiated object is written into MAIN.OBJ file.

Preprocessor Directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The C++Builder preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them. C++Builder supports these preprocessor directives:

# (null directive)	#ifdef
#define	#ifndef
#elif	#include
#else	#line
#endif	#pragma
#error	#undef
#if	

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

# (null directive)

Directives

Syntax

```
#
```

Description

The null directive consists of a line containing the single character #. This line is always ignored.

#define

See also Example Directives

Syntax

```
#define macro_identifier <token_sequence>
```

Description

The **#define** directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Each occurrence of *macro_identifier* in your source code following this control line will be replaced in the original position with the possibly empty *token_sequence* (there are some exceptions, which are noted later). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

An empty token sequence results in the removal of each affected macro identifier from the source code.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of *nested macros*: The expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a directive will not be recognized by the preprocessor. There are these restrictions to macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro won't be expanded during its own expansion (so `#define A A` won't expand indefinitely).

Example

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
#define GETSTD #include <stdio.h>
```

#undef

See also Example Directives

Syntax

```
#undef macro_identifier
```

Description

You can undefine a macro using the **#undef** directive. **#undef** detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined. No macro expansion occurs within **#undef** lines.

The state of being *defined* or *undefined* turns out to be an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is *exactly* the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

Example

```
#define BLOCK_SIZE 512
  .
  .
  .
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be illegal "unknown" identifier */
  .
  .
  .
#define BLOCK_SIZE 128   /* redefinition */
```

Using the -D and -U command-line options

See also

Identifiers can be defined and undefined using the command-line compiler options -D and -U.

The command line

```
BCC32 -Ddebug=1; paradox=0; X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
#define X
#undef mysym
```

in the program.

Keywords and Protected Words as Macros

It is legal but ill-advised to use C++Builder keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long    /* legal and possibly useful */
```

The following predefined global identifiers *cannot* appear immediately following a #define or #undef directive:

_ _DATE_ _	_ _FILE_ _	_ _LINE_ _
_ _STDC_ _	_ _TIME_ _	

Macros with Parameters

See also

The following syntax is used to define a macro with parameters:

`#`**`define`** *`macro_identifier`*`(<`*`arg_list`*`>)` *`token_sequence`*

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note there can be no whitespace between the macro identifier and the **(**. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *placeholder*.

Such macros are called by writing

*`macro_identifier<whitespace>`***`(`***`<actual_arg_list>`***`)`**

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg_list* of the #define line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Nesting Parentheses and Commas

The *actual_arg_list* can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.

```
#define ERRMSG(x, str) showerr("Error:, x. str)
#define SUM(x,y)  ((x) + (y))
.
.
.
ERRMSG(2, "Press Enter, then Esc");
/* expands to:  showerr("Error",2,"Press Enter, then Esc");
return SUM(f(i, j), g(k,l));
/* expands to: return ((f(i,j) + (g(k,l))); */
```

Token Pasting with

You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers.

Given the definition

```
#define VAR(i, j) (i##j)
```

the call `VAR(x, 6)` expands to `(x6)`. This replaces the older nonportable method of using `(i/**/j)`.

Converting to Strings with

The # symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement.

Given the following definition:

```
  #define TRACE(flag) printf(#flag "=%d\n", flag)
```

the code fragment

```
  int highval = 1024;
  TRACE(highval);
```

becomes

```
  int highval = 1024;
  printf("highval" "=%d\n", highval);
```

which, in turn, is treated as

```
  int highval = 1024;
  printf("highval=%d\n", highval);
```

Using the Backslash (\) for Line Continuation

A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.

```
#define WARN "This is really a single-\
line warning."
.
.
.
puts(WARN)
/* Screen will show: This is really a single-line warning. */
```

Side Effects and Other Dangers

The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once.

Compare *CUBE* and *cube* in the following example.

```
int cube(int x) {
   return x* x*x;
   }
#define CUBE(x)  ( (x)* (x) * (x) )
...
int b = 0, a = 3;
b = cube(a++);
/* cube() is passed actual arg = 3; so b = 27; a now = 4 */
a = 3;
b = CUBE(a++);
/* expands as ((a++)*(a++)*(a++)); a now = 6 */
```

#include

Example Directives

Syntax

```
#include <header_name>
#include "header_name"
#include macro_identifier
```

Description

The **#include** directive pulls in other named files, known as *include files*, *header files*, or *headers*, into the source code. The syntax has three versions:

- The first and second versions imply that no macro expansion will be attempted; in other words, *header_name* is never scanned for macro identifiers. *header_name* must be a valid file name with an extension (traditionally .h for header) and optional path name and path delimiters.
- The third version assumes that neither **<** nor " appears as the first non-whitespace character following **#include**; further, it assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <*header_name*> or "*header_name*" formats.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *header_name*, only that directory will be searched.

The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file.

Header File Search with <header_name>

The <*header_name*> version specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

Header File Search with "header_name"

The "*header_name*" version specifies a user-supplied include file; the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the include directories as in the <*header_name*> situation.

Example

This **#include** statement causes it to look for stdio.h in the standard include directory.

```
#include <stdio.h>
```

This **#include** statement causes it to look for MYINCLUD.H in the current directory, then in the default directories.

```
#include "myinclud.h"
```

After expansion, this **#include** statement causes the preprocessor to look in C:\BCB\INCLUDE\MYSTUFF.H and nowhere else.

```
#define myinclud "C:\BCB\INCLUDE\MYSTUFF.H"
/* Note: Single backslashes OK here; within a C statement you would
   need "C:\BCB\INCLUDE\\MYSTUFF.H" */
#include  myinclud
/* macro expansion */
```

Conditional compilation

C++Builder supports conditional compilation by replacing the appropriate source-code lines with a blank line. The linees thus ignored are those beginning with **#** (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

defined

See also Keywords

Syntax

```
#if defined[(] <identifier> [)]
#elif defined[(] <identifier> [)]
```

Description

Use the **defined** operator to test if an identifier was previously defined using #define. The **defined** operator is only valid in #if and #elif expressions.

Defined evaluates to 1 (true) if a previously defined symbol has not been undefined (using #undef); otherwise, it evaluates to 0 (false).

Defined performs the same function as #ifdef.

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use defined repeatedly in a complex expression following the #if directive; for example,

```
#if defined(mysym) && !defined(yoursym)
```

#if, #elif, #else, and #endif

See also Directives

Syntax

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
   .
   .
   .
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
```

Description

C++Builder supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The conditional directives #**if**, #**elif**, #**else**, and #**endif** work like the normal C conditional operators. If the *constant-expression-1* (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by *section-1*, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the C++Builder compiler. Otherwise, if *constant-expression-1* evaluates to zero (false), *section-1* is ignored (no macro expansion and no compilation).

In the *true* case, after *section-1* has been preprocessed, control passes to the matching #**endif** (which ends this conditional sequence) and continues with *next-section*. In the *false case*, control passes to the next #**elif** line (if any) where constant-expression-2 is evaluated. If true, *section-2* is processed, after which control moves on to the matching #**endif**. Otherwise, if *constant-expression-2* is false, control passes to the next #**elif**, and so on, until either #**else** or #**endif** is reached. The optional #**else** is used as an alternative condition for which all previous tests have proved false. The #**endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each #**if** must be matched with a closing #**endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each #**if** can be matched with its correct #**endif**.

The constant expressions to be tested must evaluate to a constant integral value.

#ifdef and #ifndef

See also Directives

Syntax

```
#ifdef identifier
#ifndef identifier
```

Description

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous #define command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if *identifier* is currently defined, and the same effect as

```
#if 0
```

if *identifier* is currently undefined.

#ifndef tests true for the "not-defined" condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if *identifier* is currently defined, and the same effect as

```
#if 1
```

if *identifier* is currently undefined.

The syntax thereafter follows that of the #if, #elif, #else, and #endif.

An identifier defined as NULL is considered to be defined.

#line

Directives

Syntax

```
#line integer_constant  <"filename">
```

Description

You can use the **#line** directive to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program.

The #**line** directive indicates that the following source line originally came from line number *integer_constant* of *filename*. Once the *filename* has been registered, subsequent #**line** commands relating to that file can omit the explicit *filename* argument.

Macros are expanded in #**line** arguments as they are in the #include directive.

The #**line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

#error

Example Directives

Syntax

```
#error errmsg
```

Description

The **#error** directive generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional statement that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional statement that is true for the undesired case.

Example

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

#pragma summary

Directives

Syntax

```
#pragma directive-name
```

Description

With #**pragma**, C++Builder can define the directives it wants without interfering with other compilers that support #**pragma**. If the compiler doesn't recognize *directive-name*, it ignores the #**pragma** directive without any error or warning message.

C++Builder supports the following #**pragma** directives:

#pragma anon_struct

#pragma argsused

#pragma codeseg

#pragma comment

#pragma exit

#pragma hdrfile

#pragma hdrstop

#pragma inline

#pragma intrinsic

#pragma link

#pragma message

#pragma option

#pragma resource

#pragma startup

#pragma warn

#pragma anon_struct

See also #pragma

Syntax

```
#pragma anon_struct on
#pragma anon_struct off
```

Description

The **anon_struct** directive allows you to compile anonymous structures embedded in classes.

```
#pragma anon_struct on
struct S {
   int i;
   struct {  // Embedded anonymous struct
      int   j ;
      float x ;
   };
   class {  // Embedded anonymous class
   public:
      long double ld;
   };
S() { i = 1; j = 2; x = 3.3; ld = 12345.5;}
};
#pragma anon_struct off

void main() {
   S mystruct;
   mystruct.x = 1.2;  // Assign to embedded data.
   }
```

#pragma argsused

See also #pragma

Syntax

```
#pragma argused
```

Description

The **argsused** pragma is allowed only between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

#pragma codeseg

See also #pragma

Syntax

```
#pragma codeseg <seg_name> <"seg_class"> <group>
```

Description

The **codeseg** directive lets you name the segment, class, or group where functions are allocated. If the pragma is used without any of its options, the default code segment is used for function allocation.

#pragma comment

See also #pragma

Syntax

```
#pragma comment (comment type, "string")
```

Description

The **comment** directive lets you write a comment record into an output file. The *comment type* can be one of the following values:

Value	Explanation
exestr	The linker writes *string* into an .OBJ file. Your specified *string* is placed in the executable file. Such a string is never loaded into memory but can be found in the executable file by use of a suitable file search utility.
lib	Writes a comment record into an .OBJ file. The comment record is used by the linker as a library-search directory. A library module that is not specified in the linker's response-file can be specified by the **comment** LIB directive. The linker includes the library module name specified in *string* as the last library. Multiple modules can be named and linked in the order in which they are named.
user	The compiler writes *string* into the .OBJ file. The specified string is ignored by the linker.

#pragma exit and #pragma startup

See also #pragma

Syntax

```
#pragma startup function-name <priority>
#pragma exit function-name <priority>
```

Description

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the *main* function is called), or program exit (just before the program terminates through **_exit**).

The specified *function-name* must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as:

```
void func(void);
```

The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100.

Note: Priorities from 0 to 63 are used by the C libraries, and should not be used by the user.

#pragma hdrfile

See also #pragma

Syntax

```
#pragma hdrfile "filename.CSM"
```

Description

This directive sets the name of the file in which to store precompiled headers.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option -H=filename or Use Precompiled Headers to change the name of the file used to store precompiled headers.

#pragma hdrstop

See also #pragma

Syntax

```
#pragma hdrstop
```

Description

This directive terminates the list of header files eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers.

Use this pragma directive only in source files. The pragma has no effect when it is used in a header file.

#pragma inline

See also #pragma

Syntax

```
#pragma inline
```

Description

This directive is equivalent to the **-B** command-line compiler option or the IDE inline option.

This is best placed at the top of the file, because the compiler restarts itself with the **-B** option when it encounters #**pragma inline**.

#pragma intrinsic

See also Example #pragma

Syntax

```
#pragma intrinsic [-]function-name
```

Description

Use **#pragma intrinsic** to override command-line switches or IDE options to control the inlining of functions.

When inlining an intrinsic function, always include a prototype for that function before using it.

Example

This example causes the compiler to generate code for *strcpy* in your function:

```
#pragma intrinsic strcpy
```

while this version prevents the compiler from inlining *strcpy*:

```
#pragma intrinsic -strcpy
```

#pragma link

See also #pragma

Syntax

```
#pragma link “[path]modulename[.ext]”
```

Description

This directive is automatically written by the IDE and the DCC32 compiler. The directive instructs the linker to link the file into an executable file.

By default, the linker searches for *modulename* in the local directory and any path specified by the **-L** option. You can use the *path* argument to specify a directory.

By default, the linker assumes a `.obj` extension.

#pragma message

See also Example #pragma

Syntax

```
#pragma message ("text" ["text"["text" ...]])
#pragma message text
```

Description

Use #**pragma message** to specify a user-defined message within your program code.

The first form requires that the text consist of one or more string constants, and the message must be enclosed in parentheses. (This form is compatible with MSC.) The second form uses the text following the **#pragma** for the text of the warning message. With both forms of the **#pragma**, any macro references are expanded before the message is displayed.

Display of user-defined messages is on by default and can be turned on or off with the Show Warnings. This option corresponds to the compiler's **-wmsg** switch.

Example

The following example displays either "You are compiling using version xxx of C++Builder" (where xxx is the version number) or "Sorry, you are not using the C++Builder compiler".

```
#ifdef __BORLANDC__
#pragma message You are compiling using version __BORLANDC__ of C++Builder.
#else
#pragma message ("Sorry, you are not using the C++Builder compiler")
#endif
```

#pragma option

See also #pragma

Syntax

```
#pragma option [options...]
```

Description

Use #**pragma option** to include command-line options within your program code.

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, then return it to its default, without having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include:

-B	**-c**	**-d*name***
-D*name=string*	**-e*filename***	**-E**
-Fx	**-h**	**-l*filename***
-lexset	**-M**	**-o**
-P	**-Q**	**-S**
-T	**-U*name***	**-V**
-X	**-Y**	

You can use #**pragmas**, #**includes**, #**define**, and some #**ifs** in the following cases:

- Before the use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an #**if**, #**ifdef**, #**ifndef** or #**elif** directive.
- Before the occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can appear only in a #**pragma option** command before these events. These options are:

-E*filename*	**-f**	**-i**#
-m*	**-n***path*	**-o*filename***
-u	**-W**	**-z**

*

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

-1	**-h**	**-r**
-2	**-k**	**-rd**
-a	**-N**	**-v**
-ff	**-O**	**-y**
-G	**-p**	**-Z**

The following options can be changed at any time and take effect immediately:

-A	**-g*n***	**-zE**
-b	**-j*n***	**-zF**
-C	**-K**	**-zH**
-d	**-w*xxx***	

The options can appear followed by a dot (.) to reset the option to its command-line state.

#pragma resource

See also #pragma

Syntax

```
#pragma resource "*.dfm"
```

Description

This pragma causes the file to be marked as a form unit and requires matching .dfm and header files. All such files are managed by the IDE.

If your form requires any variables, they must be declared immediately after the pragma resource is used. The declarations must be of the form

```
   TFormName *Formname;
```

#pragma warn

See also Example #pragma

Syntax

```
#pragma warn [+|-|.]www
```

Description

The **warn** pragma lets you override specific ***-wxx****x* command-line options or check Display Warningsl in the Messages options.

Example

If your source code contains the directives:

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

the *xxx* warning will be turned on, the *yyy* warning will be turned off, and the *zzz* warning will be restored to the value it had when compilation of the file began. See the command-line options summary for a complete list of the three-letter abbreviations and the warnings to which they apply.

Predefined macros

See also

C++Builder predefines certain global identifiers known as manifest constants. Most global indentifers begin and end with two underscores (__).

Note: For readability, underscores are often separated by a single blank space. In your source code, you should never insert whitespace between underscores.

Macro	Value	Description
_ _BCOPT_ _	1	Defined in any compiler that has an optimizer.
_ _BCPLUSPLUS_ _	0x520	Defined if you've selected C++ compilation; will increase in later releases.
_ _BORLANDC_ _	0x520	Version number.
_ _CDECL_ _	1	Defined if Calling Convention is set to cdecl; otherwise undefined.
_CHAR_UNSIGNED	1	Defined by default indicating that the default **char** is **unsigned char**. Use the **-K** option to undefine this macro.
_ _CONSOLE_ _		When defined, the macro indicates that the program is a console application.
_CPPUNWIND	1	Enable stack unwinding.This is true by default; use **-xd-** to disable.
_ _cplusplus	1	Defined if in C++ mode; otherwise, undefined.
_ _DATE_ _	String literal	Date when processing began on the current file.
_ _DLL_ _	1	Defined whenever the **-WD** option is used; otherwise undefined.
_ _FILE_ _	String literal	Name of the current file being processed.
_ _LINE_ _	Decimal constant	Number of the current source file line being processed.
_M_IX86	1	Always defined. The default value is 300. You can change the value to 400 or 500 by using the **/4** or **/5** options.
_ _MSDOS_ _	1	Integer constant.
_ _MT_ _	1	Defined only if the **-WM** option is used. It specifies that the multithread library is to be linked.
_ _PASCAL_ _	1	Defined if Calling Convention is set to Pascal; otherwise undefined.
_ _STDC_ _	1	Defined if you compile with the **-A** option; otherwise, undefined.
_ _TCPLUSPLUS_ _	0x520	Version number.
_ _TEMPLATES_ _	1	Defined as 1 for C++ files (meaning that templates are supported); otherwise, it is undefined.
_ _TIME_ _	String literal	Time when processing began on the current file.
_ _TLS_ _	1	Thread Local Storage. Always true in C++Builder.
_ _TURBOC_ _	0x520	Will increase in later releases.
_WCHAR_T	1	Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.
_WCHAR_T_DEFINED		1 Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.

_Windows		Defined for Windows-only code.
_ _WIN32_ _	1	Defined for console and GUI applications.

Note: _ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _STDC_ _, and _ _TIME_ _**cannot** appear immediately following a #define or #undef directive.

C++ Exception Handling

See also

These topics describe the C++ error-handling mechanism generally referred to as exception handling. The C++Builder implementation is consistent with the proposed ANSI specification.

The exception handling mechanisms in C++Builder provide support for handling exceptions that can occur during Delphi VCL usage. As part of the Delphi support, C++Builder also provides support for handling operating system exceptions.

- Throwing an Exception
- Handling an Exception
- Exception Specifications
- Constructors and Destructors in Exception Handling
- Unhandled Exceptions
- Setting Exception Handling Options

The C++ language defines a standard for exception handling. The standard insures that the power of object-oriented design is supported throughout your program.

In accordance with the ANSI/ISO working paper specification, C++Builder supports the termination exception-handling model. When an abnormal situation arises at runtime, the program should terminate. However, throwing an exception allows you to gather information at the throw point that could be useful in diagnosing the causes which led to failure. You can also specify in the exception handler the actions to be taken before the program terminates. Only synchronous exceptions are handled, meaning that the cause of failure is generated from within the program. An event such as Control-C (which is generated from outside the program) is not considered to be an exception.

C++ exceptions can only be handled in a **try**/**catch** construct.

Syntax for **try**/**catch** construct:

try-block:

try *compound-statement handler-list*

handler-list:

handler handler-listopt

handler:

catch (*exception-declaration*) *compound-statement*

exception declaration:

type-specifier-list declarator

type-specifier-list abstract-declarator

type-specifier-list

…

throw-expression:

throw *assignment-expressionopt*

Note: The **catch** and **throw** keywords are not allowed in a C program.

The *try-block* is a statement that specifies the flow of control as the program executes. The *try-block* is designated by the **try** keyword. Braces after the keyword are used to surround a program block that can generate exceptions. The language structure specifies that any exceptions that occur should be raised within a *try-block*.

The handler is a block of code designed to handle an exception. The C++ language requires that at least one handler be available immediately after the *try-block*. There should be a handler for each exception that the program can generate.

When the program encounters an abnormal situation for which it is not designed, you may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.

The exception-handling mechanism requires the use of three keywords: try, catch, and throw. The *try-block* specified by **try** must be followed immediately by the *handler* specified by **catch**. If an exception is thrown in the *try-block*, program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so could result in abnormal termination of program.

Exception declarations

Although C++ allows an exception to be of any type, it is useful to make exceptions objects. The exception object is treated exactly the way any object would. An exception carries information from the point where the exception is thrown to the point where the exception is caught. This is information that the program user will want to know when the program encounters some anomaly at runtime.

Throwing an Exception

See also C++ Exception Handling

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is tranferred to the handler

If no handler is found, the program will call the *terminate* function. If no exceptions are thrown, the program executes in the normal fashion.

A throw expression is also referred to as a throw-point. You can specify whether an exception may be thrown by using one of the following syntax specifications:

Example 1

```
throw throw_object;
```

This example specifies that *throw_object* is to be passed to a handler.

Example 2

```
throw;
```

This example simply specifies that the last exception thrown is to be thrown again. An exception must currently exist. Otherwise, the *terminate* function is called.

Example 3

```
void my_func1() throw (A, B)
{
    // Body of function.
}
```

This example specifies a list of exceptions that *my_func1* can throw. No other exceptions will propagate out of *my_func1*. If an exception other than *A* or *B* is generated within *my_func1*, it is considered to be an unexpected exception and program control will be transferred to the *unexpected* function.

Example 4

```
void my_func2() throw ()
{
     // Body of this function.
}
```

The final case specifies that *my_func2* should not throw any exceptions. If any function (for example, **operator new**) in the body of *my_func2* throws an exception, such an exception should be caught and handled within the body of *my_func2*. Otherwise, such an exception is a violation of the exception specification for *my_func2*. The *unexpected* function is then called.

When an exception occurs, the throw expression initializes a temporary object of the type **T** (to match the type of argument *arg*) used in *throw*(**T** *arg*). Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

Handling an Exception

See also Examples C++ Exception Handling

The exception handler is indicated by the **catch** keyword. The handler must be placed immediately after the try-block. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call *terminate*.

Exception handlers are evaluated in the order that they are encountered. An exception is caught when its type matches the type in the **catch** statement. Once a type match is made, program control is transferred to the handler. The stack will have been unwound upon entering the handler. The handler specifies what actions should be taken to deal with the program anomaly.

A **goto** statement can be used to transfer program control out of a handler but such a statement can never be used to enter a handler.

After the handler has executed, the program can continue at the point after the last handler for the current try-block. No other handlers are evaluated for the current exception.

Example 1

```
try {
    // Include any code that might throw an exception
}
catch (T X) // Provide a handler for each exception that might be thrown above
{
   // Take some actions
}
```

This example is specifically defined to handle an object of type **T. I**f the argument is **T, T&, const T**, or **const T&**, the handler will accept an object of type **X** if any of the following are true:

- **T** and **X** are of the same type
- **T** is an accessible base class for **X** in the throw expression
- **T** is a pointer type and **X** is a pointer type that can be converted to **T** by a standard pointer conversion in the throw expression

Example 2

```
try {
    // Include any code that might throw an exception
}
catch ( ... )
{
  // Take some actions
}
```

The statement **catch** (...) will handle any exception, regardless of type. This statement, if used, must be the last handler for its *try-block*.

Exception Specifications

See also Examples C++ Exception Handling

The C++ language makes it possible for you to specify any exceptions that a function can throw. This *exception specification* can be used as a suffix to the function declaration. The syntax for exception specification is as follows:

exception-specification:

throw (*type-id-list*opt)

type-id-list:

type-id

type-id-list, type-id

The function suffix is not considered to be part of the function's type. Consequently, a pointer to a function is not affected by the function's exception specification. Such a pointer checks only the function's return and argument types. Therefore, the following is legal:

```
void f2(void) throw();          // Should not throw exceptions
void f3(void) throw (BETA); // Should only throw BETA objects
void (* fptr)();                    // Pointer to a function returning void
fptr = f2;
fptr = f3;
```

Extreme care should be taken when overriding virtual functions. Again, because the exception specification is not considered part of the function type, it is possible to violate the program design.

Example 1

In the following example, the derived class *BETA::vfunc* is defined so that it should not throw any exceptions--a departure from the original function declaration.

```
class ALPHA {
public:
    struct ALPHA_ERR {};
    virtual void vfunc(void) throw (ALPHA_ERR) {}; // Exception specification
};

class BETA : public ALPHA {
    void vfunc(void) throw() {};    // Exception specification is changed
};
```

The following are examples of functions with exception specifications.

```
void f1();                   // The function can throw any exception

void f2() throw();           // Should not throw any exceptions

void f3() throw( A, B* ); // Can throw exceptions publicly derived from A,
                             // or a pointer to publicly derived B
```

The definition and all declarations of such a function must have an exception specification containing the same set of *type-id*'s. If a function throws an exception not listed in its specification, the program will call *unexpected*. This is a runtime issue--it will not be flagged at compile time. Therefore, care must be taken to handle any exceptions which can be thrown by elements called within a function.

Example 2

```
// HOW TO MAKE EXCEPTION-SPECIFICATIONS AND HANDLE ALL EXCEPTIONS
#include <iostream.h>

// EXCEPTION DECLARATIONS
class Alpha {
    // Include something that shows why you chose to throw this exception.
};
Alpha alpha_inst;

class Beta {
    // Include something that shows why you chose to throw this exception.
};
Beta beta_inst;

// THROW ONLY Alpha OR Beta TYPE OBJECTS
void f3(char c) throw (Alpha, Beta) {
    cout << "f3() was called" << endl;
    if (c == 'a')
        throw( alpha_inst );
    if (c == 'b')
        throw( beta_inst );
    else ; // DO NOTHING WITH OTHER CHARACTERS
    }

// SHOULD NOT THROW EXCEPTIONS
void f2(char ch) throw() {
    try {                          // WRAP ALL CODE IN A TRY-BLOCK
        cout << "f2() was called" << endl;
        f3(ch);
        }
    // HERE ARE HANDLERS FOR THE EXCEPTIONS WE KNOW COULD BE THROWN
    catch (Alpha& alpha_inst) { cout << "Caught Alpha exception.";}
    catch (Beta& beta_inst) { cout << "Caught Beta exception.";}

    // IF THE CODE IS MODIFIED LATER SO THAT SOME
    // OTHER EXCEPTION IS THROWN, IT IS HANDLED HERE
    // AND WE AVOID VIOLATING THE f2() THROW SPECIFICATION
    catch ( ... ) {
        // BUT, WE POST OURSELVES A WARNING MESSAGE.
        cout << "Warning: f2() has elements with exceptions!" << endl;
        }
    }

int main(void) {
    char trigger;

    try {
        cout << "Input a character:";
        cin >> trigger;
        f2(trigger);
        cout << "\nSuccess.";
        return 0;   // WE GET HERE ONLY IF EVERYTHING EXECUTES WELL.
        }
    catch ( ... ) {
        cout << "Need more handlers!";
        return 1;
        }
    }
```

Sample output when ‘a’ is the input:

```
Input a character: a

f2() was called

f3() was called

Caught Alpha exception.

Success.
```

Example 3

If an exception is thrown which is not listed in the exception specification, the unexpected function will be called.

The following examples illustrate the different sequence of events that can occur when *unexpected* is called. The behavior depends on whether you register a function with *set_unexpected()* or *set_terminate()*.

Program behavior when a function is registered with set_unexpected():

```
unexpected()   // CALLED AUTOMATICALLY
   |
   |
   |                      // DEFINE YOUR UNEXPECTED HANDLER
   |                     unexpected_function my_unexpected( void )
   |                     {
   |                         // DEFINE ACTIONS TO TAKE.
   |                         // POSSIBLY MAKE ADJUSTMENTS.
   |                     }
   |
   |                     // Now,register you handler
   |                     set_unexpected( my_unexpected );
   |
my_unexpected();
```

Program behavior when no function is registered with set_unexpected() but there is a function registered with set_terminate():

```
unexpected()   // CALLED AUTOMATICALLY
   |
terminate()
   |
   |               // DEFINE YOUR TERMINATION SCHEME
   |               terminate_function my_terminate( void )
   |                   {
   |                   // TAKE ACTIONS BEFORE TERMINATING
   |                   // SHOULD NOT THROW EXCEPTIONS
   |                   exit(1); // MUST END SOMEHOW.
   |                   }
   |
   |               // REGISTER YOUR TERMINATION FUNCTION
   |               set_terminate( my_terminate )
   |
   |
my_terminate()
// PROGRAM ENDS.
```

Constructors and Destructors in Exception Handling

See also C++ Exception Handling

When an exception is thrown, the copy constructor is called for the exception. The copy constructor is used to initialize a temporary object at the throw point. Other copies may be generated by the program.

When program flow is interrupted by an exception, destructors are called for all automatic objects which were constructed since the beginning of the the try-block was entered. If the exception was thrown during construction of some object, destructors will be called only for those objects which were fully constructed. For example, if an array of objects was under contruction when an exception was thrown, destructors will be called only for the array elements which were already fully constructed.

The effect of calling destructors for automatic objects is referred to as stack unwinding. Stack unwinding always occurs. Destructors are called by default but the default can be switched off by using the **-xd** compiler option.

Unhandled Exceptions

See also C++ Exception Handling

If an exception is thrown and no handler is found it, the program will call the *terminate* function. This example illustrates the series of events that can occur when the program encounters an exception for which no handler can be found.

```
terminate();
   .
   .
   .
abort();   // PROGRAM ENDS.
```

C-Based Structured Exceptions

See also

C++Builder provides support for program development that makes use of structured exceptions. You can compile and link a C source file that contains an implementation of structured exceptions. In a C program, the ANSI-compatible keywords used to implement structured exceptions are _ _*except*, _ _finally, and _ _try.

Note: The **_ _finally** and **_ _try** keywords can appear only in C programs.

try-except Exception-Handling Syntax

For *try-except* exception-handling implementations the syntax is as follows:

try-block:

_ _try *compound-statement* (in a C module)

try *compound-statement* (in a C++ module)

handler:

_ _except (*expression*) *compound-statement*

try-finally Termination Syntax

For try-finally termination implementations the syntax is as follows:

try-block:

_ _try *compound-statement*

termination:

_ _finally *compound-statement*

See your Win32 documentation for additional details on the implementation of structured exceptions.

Using C-Based Exceptions in C++ Programs

See also Example

C++Builder allows substantial interaction between Delphi, C, and C++ error handling mechanisms. The following interactions are supported between C and C++:

- C structured exceptions can be used in C++ programs.
- C++ exceptions cannot be used in a C program because C++ exceptions require that their handler be specified by the **catch** keyword and **catch** is not allowed in a C program.
- An exception generated by a call to the *RaiseException* function is handled by a **try**/**_ _except** or **_ _try**/**_ _except** block. All handlers of **try**/**catch** blocks are ignored when *RaiseException* is called.
- The following C exception helper functions can be used in a C and C++ programs:
- *GetExceptionCode*
- *GetExceptionInformation*
- *SetUnhandledExceptionFilter*
- *UnhandledExceptionFilter*

C++Builder does not enforce the use of *UnhandledExceptionFilter* function only in the except filter of **_ _try**/**_ _except** or **try**/**_ _except** blocks. However, program behavior is undefined when this function is called outside of the **_ _try**/**_except** or **try**/**_ _except** block.

Handling C-based exceptions

See also Example

The full functionality of an **_ _except** block is allowed in C++. If an exception is generated in a C module, it is possible to provide a handler-block in a separate calling C++ module. If no handler is found in the calling module, the default action is to terminate the program.

If a handler can be found for the generated structured exception, the following actions can be taken:

- execute the actions specified by the handler
- ignore the generated exception and resume program execution
- continue the search for some other handler (regenerate the exception)

These actions are consistent with the design of structured exceptions.

The **_ _try**/**_ _finally** ensures that the code in the **_ _finally** block is executed no matter how the flow within the **_ _try** exits. The **_ _finally** keyword is not allowed in a C++ program and the **_ _try**/**_ _finally** block is not supported in a C++ program.

Even though the **_ _try**/**_ _finally** block is not supported in a C++ program, a C-based exception generated by the operating system or the program can still result in proper stack unwinding by using local objects within destructors. Any module compiled with the **-xd** compiler option will have destructors invoked for all objects with **auto** storage. Stack unwinding occurs from the point where the exception is thrown to the point where the exception is caught.

C-Based Exceptions in C++ Programs Example

```
/* In PROG.C */
void func(void) {

    .
    .
    .
    /* generate an exception */
    RaiseException( /* specifiy your arguments */ );

    .
    .
    .
}

// In CALLER.CPP
// How to test for C++ or C-based exceptions.
#include <excpt.h>
#include <iostream.h>

int main(void) {
    try
    {                // test for C++ exceptions
        try
        {            // test for structured exceptions
            func();
        }
        __except( /* filter-expression */ )
        {
        cout << "A structured exception was generated.";

    .
    .
    .
        /* specify actions to take for this structured exception */
        return -1;
        }
        return 0;
    }
    catch ( ... )
    {
    // handler for any C++ exception
    cout << "A C++ exception was thrown.";
    return 1;
    }
}
```

The property-method-event Model

C++Builder applications are based on the property-method-event (PME) model of programming. Applications are built using discrete software components. These components have properties that define the component's state. The properties can be changed to affect the components. The components also have built in methods, or member functions, which can be used to manipulate the component. Components trigger events when changes occur in the component. These events are passed by calling special event-handling properties set by the code using the component.

C++Builder's Visual Component Library (VCL) now allows C++ programmers to do visually what they have traditionally done by hand coding classes or by coding using an application framework such as Microsoft Foundation Class (MFC) or ObjectWindows Library (OWL).

Most of the pieces that make up a Win32 application are encapsulated in the VCL library. These pieces include:

- User-interface components, including windows, controls, menus and common dialogs.
- Database management and manipulation components for any database from a local dBASE® table to an Oracle database on an IBM mainframe.
- Windows specific components for dealing with such tasks as registry manipulation, printing and multimedia.
- Support for advanced Windows features such as multiple threads, OLE automation and timers.
- Support for classic data structures such as collections and lists.

Because all these features are now available through the VCL, and most can be manipulated visually through the C++Builder user interface, C++ programmers no longer need to create or manipulate these objects through code. Many C++Builder applications can be created by designing the application visually using the C++Builder Form Designer and adding a few lines of code to the key component's event handlers. Remember this rule of thumb: use the VCL objects whenever possible and resist the urge to write new code until all other possibilities have been exhausted.

Components vs. classes

At first glance, C++Builder's components appear to be just like any other C++ class. But there are differences between components in C++Builder and the standard C++ class hierarchies most classic C++ programmers work with. Some differences are:

- All C++Builder components are descended from TComponent.
- Components are usually used as is, and manipulated through their properties, rather than serving as "base classes" to be sub-classsed to add or change functionality. When a component is inherited, it is usually to add specific code to already existing event handling member functions.
- VCL Components can only be allocated on the heap, not on the stack (that is, they *must* be created with the **new** operator).
- Properties of components intrinsically contain runtime type information.
- Components can be added to the Component Palette in the C++Builder user interface and manipulated via the C++Builder Form Designer.

Components often achieve a better degree of encapsulation than is usually found in standard C++ classes. For example, take the case of a dialog containing a push button. When a user clicks on the button in a Windows C++ program, a WM_LBUTTONDOWN message is generated by the system. This message must be caught by the programmer, typically either in a switch statement or a message map or response table, and dispatched to the correct routine that should be run in response to that message. For example, an OWL application uses the following macro to associate an EV_BN_CLICKED event (a WM_LBUTTONDOWN message) generated by the IDEVENTBUTTON button on a TEventTestDlgClient dialog with the EventBNClicked( ) function in response.

```
DEFINE_RESPONSE_TABLE1(TEventTestDlgClient, TDialog)
//{{TEventTestDlgClientRSP_TBL_BEGIN}}
  EV_BN_CLICKED(IDEVENTBUTTON, EventBNClicked),
//{{TEventTestDlgClientRSP_TBL_END}}
END_RESPONSE_TABLE;
```

In C++Builder, the push button component is pre-programmed to respond to a mouse click using its built in *OnClick* event handler. The programmer does not have to provide any means to process that a button has been clicked. The programmer only provides the routine that will be called when the button is clicked, and through the Object Inspector of the C++Builder Form Designer, assigns that routine to the *OnClick* event handler of the button.

Properties vs. setter/getter functions

Properties are a C++ language extension in C++Builder that enhance the functionality of the getter and/or setter functions that are usually part of C++ classes. For example, classes that were declared as follows:

```
class Foo {
     int howMany;
  public:
     Foo() {howMany = 0;};
     void setValue(int n) {howMany = n;};
     int getValue() {return howMany};
};
```

can now be declared as:

```
class Foo {
     int howMany;
     void setValue(int n) {howMany = n;};
     int getValue() {return howMany;};
  public:
     Foo() {howMany = 0;};
  __published:
     __property int count = {read=getValue, write=setValue};
};
```

Both classes contain a private data member called *howMany*. Both use a getter (*getValue*) and a setter (*setValue*) function to get and set the value of *howMany*. But in the second class, the user gets and sets the value of *howMany* by assigning a value to, or reading the value of, *count*. The use of the property *count* simplifies the interface for the user of the class, while still keeping the private data member *howMany* private.

Properties allow for the creation of intelligent data members in a class. In the example above, the setter function could be rewritten as:

```
void setValue(int n) {howMany = n < 0 ? 0 : n;};
```

which prevents a value less than 0 from being assigned to *howMany*. For example:

```
void Foobar()
{
  Foo myFoo;
  myFoo.count = 10;
  int x = myFoo.count;  // 10 assigned to x;
  myFoo.count = -10;    // -10 changed to 0 on assignment
  x = myFoo.count;      // 0 assigned to x;
}
```

Working with legacy code

The C++Builder compiler can compile most Win32 C and C++ code that is compatible with Borland C++ 5.0. C++Builder cannot compile 16-bit Windows or DOS programs.

Because of C++Builder's unique exception handling mechanism, object code and library modules originally compiled using Borland C++ 5.01 or earlier must be recompiled with C++Builder's compiler before they can be linked into an C++Builder application. Also, object code modules compiled with the C++Builder compiler will not link into projects built with Borland C++ 5.01 or earlier versions.

C++Builder provides a non-VCL dependent multi-threaded runtime library (RTL) to support legacy applications. This library, called CW32MT.LIB, does not support the VCL's enhancements to catching operating system exceptions since doing so would require the use of the VCL.

In-line assembler

The C++Builder compiler does not have a built-in in-line assembler. Using the **asm** keyword (or its variants) in your C++ program code to in-line assembly language statements causes the compiler to output an assembly language source code file (.asm) of your program rather than an object code file. The .asm file can be assembled with Borland's Turbo Assember 5.0 (available separately).

Object Windows Library (OWL) and Microsoft Foundation Classes (MFC) applications

OWL and MFC do not ship with C++Builder, but OWL 5.0 and MFC 4.1 applications will compile in C++Builder. To compile OWL or MFC code, the OWL or MFC libraries must be recompiled from the source using the C++Builder compiler. OWL also requires that the CLASSLIB.LIB library be recompiled. MFC must be compiled from the sources that shipped with Borland C++ 5.01 or a later Borland supplied version.

Be certain not to overwrite the Borland C++ version of the libraries. Borland C++ and C++Builder must each have their own versions of the libraries compiled with their respective compilers.

Some OWL class names conflict with VCL class names. To avoid ambiguous declarations C++Builder uses namespaces to differentiate VCL classes.

Creating forms in memory, default behavior

VCL components created using the Borland C++Builder IDE (such as forms and controls on forms) are automatically created when the application is started, and destroyed when the application terminates. For every form in a given application that is created using the Form Designer, a global variable with the same name as the name given to the form in the Form Designer is created. This variable is a pointer to an object of the form's class and is used to reference the form while the application is running. Any source code (.cpp) file that includes the form's header (.h) file has access to the form via this pointer. Also, by default, code is included in the application's *WinMain()* function to actually create the form in memory. Whether or not the form is created when the application is started is dependent on whether the form is included in the **Auto-create forms** list on the **Forms** page of the **Project Options** dialog. Assuming the form is auto-created (the default), the form exists in memory for the duration of the run of the application, and the form can be invoked at any time by using its *Show()* or *ShowModal()* method (if it's not already visible).

Creating forms at runtime

Any object created manually from a class descended from *TObject*, such as a form, must be created on the heap using the **new** operator. For example:

```
TMyForm MyForm(NULL);                   // error
TMyForm *MyForm = new TMyForm(NULL);   // ok
```

Users are discouraged from overloading the **new** operator for VCL based classes.

If the default global variable for the form, that is created by the Form Designer, is to be used to access the form created via **new**, be certain that the form is not auto-created when the application is initialized or memory will be wasted and the auto-created form will be inaccessible. For example, consider the following sample application. The application contains two forms: *MainMForm*, which is auto-created and displayed when the application starts, and *ResultsForm*, which is created (but not displayed) when the application starts. The project source file for the application appears below:

```
#include <vcl\vcl.h>
#pragma hdrstop
//---------------------------------------------------------------------------
USEFORM("MainForm.cpp", MainMForm);
USERES("ParamTest.res");
USEFORM("Results.cpp", ResultsForm);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  Application->Initialize();
  Application->CreateForm(__classid(TMainMForm), &MainMForm);
  Application->CreateForm(__classid(TResultsForm), &ResultsForm);
  Application->Run();

  return 0;
}
```

MainMForm and *ResultsForm* are pointer variables with global scope that point to their respective forms. A C++Builder programmer decides to create a modal instance of *ResultsForm* when the user presses a button on *MainMForm*. The event handler looks like this:

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  ResultsForm = new TResultsForm(this);   // BAD! Overwrites global reference!
  ResultsForm->ShowModal();
  delete ResultsForm;
}
```

In the case above, the programmer created a new instance of the *ResultsForm*, **overwriting** the reference to the auto-created instance. The auto-created instance still exists, but is now inaccessible. After the event-handler terminates, *ResultsForm* no longer points to a valid form. Any attempt to dereference *ResultsForm* will likely result in a crash of the application.

Modal forms

If the programmer can use the global instance of ResultsForm, the event handler could be written as follows:

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  ResultsForm->ShowModal();
}
```

This version displays the auto-created form without creating a new instance or destroying the existing instance when the event handler completes.

If the programmer really needs a new instance of the form created in the event handler, it can be done in a couple ways. The code in the first event handler shown above (utilizing **new**) can work correctly if *ResultsForm* is **not** auto-created when the application starts. Auto-creation can be avoided by removing *ResultsForm* from the **Auto-create forms** list on the **Forms** page of the **Project Options** dialog, or by manually removing the line:

```
Application->CreateForm(__classid(TResultsForm), &ResultsForm);
```

from *WinMain()*. The event handler in the example deletes the form after it's closed, so the form would need to be reinstantiated using **new** if *ResultForm* was needed elsewhere in the application.

A safer way to create a unique instance of the modal form would be to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not, as the code in the event handler make no reference to it. For example:

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  TResultsForm *rf = new TResultsForm(this);    // rf is local form instance
  rf->ShowModal();
  delete rf;                                     // form safely destroyed
}
```

Notice how the global instance of the form is never used in this version of the event handler.

In many applications, only the global instances of forms are used. In cases where a new modal instance of a form is required, and the use of the that form is limited to a discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Modeless forms

If the form instance being created is modeless, using a local reference variable may not be an option. The *Show()* method, which opens a form modelessly, returns as soon as the form opens. So if the event handler below were used:

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  TResultsForm *rf = new TResultsForm(this);    // rf is local form instance
  rf->Show();                                    // returns after opening form
  delete rf;                                     // BAD! Form destroyed
}
```

As soon as the form was created by *Show()* it would be destroyed by the **delete** statement. An even worse case would be if the **delete** command were not included in the event handler. Then the reference variable *rf* would go out of scope immediately after the form was created and the form object would be inaccessible.

It is necessary that reference variables for modeless forms be guaranteed to exist as long as the form is in use. This means that these variables should have global scope. In most cases, the global reference variable created by the Form Designer (the variable name that matches the name property of the form) is used. If additional instances of the form are required, separate global variables (of type pointer to the form class) should be declared.

Passing parameters to forms

Most of the time, the forms used in a C++Builder application are created using the C++Builder Form Designer. Forms created this way have a single constructor that takes one argument, *TComponent* Owner*. This argument is a pointer to the owner (the calling application object or form object) of the form being created. This argument can be NULL.

To pass additional arguments to a form you need to create a separate constructor. The form will also need to be instantiated using the **new** operator. The example form class below was created using the Form Designer. A second constructor, with the additional argument *int whichButton*, was manually added.

```
class TResultsForm : public TForm
{
__published:    // IDE-managed Components
    TLabel *ResultsLabel;
    TButton *OKButton;
    void __fastcall OKButtonClick(TObject *Sender);
private:        // User declarations
public:         // User declarations
    virtual __fastcall TResultsForm(TComponent* Owner);
    virtual __fastcall TResultsForm(int whichButton, TComponent* Owner);
};
```

Here is what a sample constructor might look like. This constructor uses the **int** parameter to set the *Caption* property of a *Label* control on the form.

```
__fastcall TResultsForm::TResultsForm(int whichButton, TComponent* Owner)
  : TForm(Owner)
{
  switch (whichButton) {
    case 1:
      ResultsLabel->Caption = "You picked the first button!";
      break;
    case 2:
      ResultsLabel->Caption = "You picked the second button!";
      break;
    case 3:
      ResultsLabel->Caption = "You picked the third button!";
  }
}
```

When creating an instance of a form with multiple constructors, select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* utilizing the extra parameter:

```
void __fastcall TMainMForm::SecondButtonClick(TObject *Sender)
{
  TResultsForm *rf = new TResultsForm(2, this);
  rf->ShowModal();
  delete rf;
}
```

Retrieving data from forms

Most real world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. To get information from a form a couple techniques can be used.

Modeless forms

Information from modal forms can be easily extracted by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modal form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red", "Green", "Blue", etc). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* everytime a new color is selected. The class declaration for the form is as follows:

```
class TColorForm : public TForm
{
  __published:    // IDE-managed Components
    TListBox *ColorListBox;
    void __fastcall ColorListBoxClick(TObject *Sender);
  private:        // User declarations
    String getColor();
    void   setColor(String);
    String curColor;
  public:         // User declarations
    virtual __fastcall TColorForm(TComponent* Owner);
    __property String CurrentColor = {read=getColor, write=setColor};
};
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The setter function *setColor* is used by *CurrentColor* to store the actual value for the property in the private data member *curColor*:

```
void __fastcall TColorForm::ColorListBoxClick(TObject *Sender)
{
  int index = ColorListBox->ItemIndex;
  if (index >= 0) {     // make sure a color is really selected
    CurrentColor = ColorListBox->Items->Strings[index];
  }
  else             // no color selected
    CurrentColor = "";
}
//----------------------------------------------------------------------
void TColorForm::setColor(String s)
{
  curColor = s;
}
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button on *ResultsForm* called *UpdateButton* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
  if (ColorForm->Name == "ColorForm") {   // verify ColorForm exists
    String s = ColorForm->CurrentColor;
    // do something with the color name string
  }
}
```

The event handler first verifies that *ColorForm* exists by checking the value of its *Name* property. It then simply gets the value of *ColorForm's CurrentColor* property. The query of *CurrentColor* calls its getter function *getColor* which is simply:

```
String TColorForm::getColor()
{
  return curColor;
}
```

If *ColorForm's getColor* function were public, then another form could get the current color without going through a property (e.g. `String s = ColorForm->getColor();`). In fact, there's nothing to prevent another form from getting *ColorForm's* currently selected color by checking the listbox selection directly:

```
String s = ColorListBox->Items->Strings[ColorListBox->ItemIndex];
```

However, the use of a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

Modal forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to know how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But what about situations where form B is deleted from memory when it's closed? Since a form does not have an explicit return value, a way to preserve important information from the form is to give it a place to store information before it's destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
class TColorForm : public TForm
{
  __published:    // IDE-managed Components
     TListBox *ColorListBox;
     TButton *SelectButton;
     TButton *CancelButton;
     void __fastcall CancelButtonClick(TObject *Sender);
     void __fastcall SelectButtonClick(TObject *Sender);
  private:        // User declarations
     String* curColor;
  public:         // User declarations
     virtual __fastcall TColorForm(TComponent* Owner);
     virtual __fastcall TColorForm(String* s, TComponent* Owner);
};
```

The form has a listbox called *ColorListBox* with a list of names of colors. There is a button called *SelectButton* which when pressed makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button which simply closes the form.

Note that there is a user defined constructor added to the class that takes a String* argument. Presumably, this String* is a string that the form launching *ColorForm* knows about. The implementation of this constructor is:

```
__fastcall TColorForm::TColorForm(String* s, TComponent* Owner)
  : TForm(Owner)
{
  curColor = s;
  *curColor = "";
}
```

which saves the pointer to the private data member *curColor* and initializes the string to an emply string. Note that to use the user-defined constructor the form must be created using the **new** operator. Make sure the form is not auto-created when the application is started if you don't need to use an auto-created version of the form.

The user is then expected to select a color from the listbox and press *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* is defined as:

```
void __fastcall TColorForm::SelectButtonClick(TObject *Sender)
{
  int index = ColorListBox->ItemIndex;
  if (index >= 0)
    *curColor = ColorListBox->Items->Strings[index];
  Close();
}
```

Notice how the event handler saves the selected color name at the string address that was passed to the constructor.

To use *ColorForm* effectively the calling form needs to pass the constructor an existing string address. For example, assume *ColorForm* was being instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked. The event handler would be as follows:

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
  String s;
  GetColor(&s);
  if (s != "") {
    // do something with the color name string
  }
  else {
    // do something else because no color was picked
  }
}
//---------------------------------------------------------------------
void TResultsForm::GetColor(String *s)
{
  ColorForm = new TColorForm(s, this);
  ColorForm->ShowModal();
  delete ColorForm;
}
```

UpdateButtonClick creates a String called *s*. The address of *s* is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to *s* as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in *s*, assuming that a color was selected. Otherwise *s* contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This simple example used one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. One thing to keep in mind is to always provide a mechanism to let the calling form know if the modal form was closed without making any changes or selections (such as having *s* default to an empty string in the example).

Using Delphi forms in C++Builder projects

See also

Borland C++Builder includes a modified version of the Borland Delphi compiler that allows Delphi units and forms to be compiled and linked into C++Builder applications.

To add a Delphi unit to your C++Builder project, use the **Add to Project** option of the **Project** menu, or select the “**+**” icon on the **Project Manager** window, to open the **Add to Project** dialog. Select **Pascal unit** as the **File Type** and select the Object Pascal (.PAS) file you wish to add to the project.

If you are copying Object Pascal source files from a Delphi project directory over to a C++Builder project directory, be sure you copy the form definition (.DFM) files and Windows resource (.RES) files associated with those source files to the C++Builder project directory as well.

When the C++Builder project containing the Delphi unit(s) is built, the Delphi units will automatically be compiled with the C++Builder version of the Delphi compiler. This compiler produces the following output:

1. A Delphi compiled unit (.DCU) file which can be linked into a Delphi project.
2. A C++ header file with an .HPP extension containing the C++ equivalents of the Object Pascal class declarations contained in the Object Pascal source file.
3. A object code file (.OBJ) in a format compatible with C++Builder. Special care is required for constructors hoisted from Object Pascal source to C++ compilable format.

The C++ header files the compiler produces for the Delphi units can be included (with #include) in any C++ source files that utilize the classes declared in the Delphi units. (In Delphi, classes are usually declared and defined in the same source file. In C++ classes are usually declared in a header file, and defined in a C++ source file.)

C++Builder supports virtually all intrinsic Delphi data types, thereby allowing C++Builder code to utilize the compiled Delphi units. Be certain to always use the **__fastcall** calling convention when calling Delphi methods from C++, unless the Delphi methods have been explicitly defined to use a different calling convention.

Hoisted constructors

Some special care might be required when Object Pascal source code is recompiled for C++ compliance. Object Pascal allows class constructors to be declared with names that don't match their class name. There could be a class *Base* with constructor *Create()*.

When the *Base* class is hoisted to C++ compliant format, its constructor must be renamed. Therefore, *Base.Create()* from Object Pascal becomes *myClass::myClass()* in a C++ source file.

If *Base* is being derived from, the hoisted constructor is added, with appropriate access, to the inheriting class. Object Pascal constructors retain their access scope. This is to say, public constructors are mapped into the public section of the inheriting class; protected constructors are mapped to protected class sections.

```
class myClass : public Base {
public:
   myClass( float f) {};
   myClass() {};        // This was Base.Create()
};
```

If *Base* had a second constructor, *Base.AnotherCreate()*, it is also renamed and hoisted into the C++ class. After renaming, this second constructor is indistinguishable from the first case. To distinguish the constructors you must add an argument.

```
class myClass : public Base {
public:
   myClass( float f) {};
   myClass() {};        // This was Base.Create()
   myClass(int =0) {};  // Requires manual fixup
};
```

If *Base* derived from *FirstBase* with its protected constructor *FirstBase.FirstBaseCreate(i Integer)*, its constructor must also be written in *myClass*. Note that *FirstBase* has an argument list that will make it indistinguishable from another *myClass* constructor.

```
class myClass : public Base {  // Base inherits from FirstBase.
public:
   myClass( float f) {};
   myClass() {};        // This was Base.Create()
   myClass(int =0) {};  // Requires manual fixup
protected:
   myClass(int i; int = 0); // Requires manual fixup

};
```

To allow you to identify the relation between old and new (that is, Object Pascal and C++) constructors, the recompilation process writes comments into the .HPP header file.

What is a dynamic-link library?

Dynamic-link libraries (DLLs) provide a way to modularize applications so that functionality can be updated and reused more easily. They also help reduce memory overhead when several applications use the same functionality at the same time, because although each application gets its own copy of the data, they can share the code.

In Windows, DLLs are modules that contain functions and data. A DLL is loaded at runtime by its calling modules (.EXE or DLL). When a DLL is loaded, it is mapped into the address space of the calling process.

DLLs can define two kinds of functions: exported and internal. The exported functions can be called by other modules. Internal functions can only be called from within the DLL where they are defined.

The creation and use of DLLs is described in detail in the Microsoft® Win32 SDK Reference.

Using DLLs in C++Builder

A Windows DLL can be used by a C++Builder application just as it would be by any C++ application.

To statically load a DLL when your C++Builder application is loaded, link the import library file for that DLL into your C++Builder application at link time. To add an import library to a C++Builder application, open the make file (.mak file) for the application and add the import library name to the library file list assigned to the ALLLIB variable. If necessary, add the path of the import library to the path(s) listed for the -L option of LFLAGS (linker options) variable. The exported functions of that DLL then become available for use by your application. Prototype the DLL functions your application uses with __declspec`(dllimport)` modifier:

```
__declspec(dllimport) return_type imported_function_name(parameters);
```

To dynamically load a DLL during the run of a C++Builder application use the Windows API function LoadLibrary() to load the DLL, then use the API function GetProcAddress() to obtain pointers to the individual functions you wish to use.

Additional information on using DLLs can be found in the Microsoft® Win32 SDK Reference.

Creating DLLs in C++Builder

There are no major differences in creating DLLs in C++Builder as opposed to standard C++. Exported functions in the code should be identified with the __declspec `(dllexport)` modifier as they would be in Borland C++ or Microsoft Visual C++. For example, the following code is legal in C++Builder and other Windows C++ compilers:

```
// MyDLL.cpp
double dblValue(double);
double halfValue(double);
extern “C” __declspec(dllexport) double changeValue(double, bool);

double dblValue(double value)
{
  return value * value;
};

double halfValue(double value)
{
  return value / 2.0;
}

double changeValue(double value, bool whichOp)
{
  return whichOp ? dblValue(value) : halfValue(value);
}
```

In the code above, the function *changeValue* is exported, and therefore made available to calling applications. The functions *dblValue* and *halfValue* are internal, and cannot be called from outside of the DLL.

In the C++Builder IDE, a new DLL project can be created by selecting File | New and selecting the DLL icon from the New tab. A new edit window opens and the project options are set to build a DLL, rather than an EXE. Notice that by default VCL.H is included in the new DLL source. If your DLL does not use any VCL components, you can remove this line.

Additional information on creating DLLs can be found in the Microsoft® Win32 SDK Reference.

Compiling DLLs

To compile and link a DLL from the C++Builder IDE, set the **Application Target** option on the **Linker** page of the **Project Options** dialog to **Generate DLL**, then compile the project normally.

If compiling from the command line, invoke the linker (either TLINK32.EXE or ILINK32.EXE) with the `-Tpd` switch. For example:

```
tlink32 /c /aa /Tpd c0d32.obj mydll.obj, mydll.dll, mdll.map, import32.lib
  cw32mt.lib
```

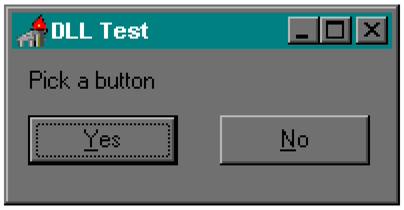
If you plan to use the DLL from within other C or C++ programs and want the calling application to automatically load the DLL when the application is run, you should use the IMPLIB.EXE command-line utility to create an import library for the DLL which can be linked into calling applications. For example:

```
implib mydll.lib mydll.dll
```

Creating DLLs containing VCL components

One of the strengths of DLLs is that a DLL created with one development tool can often be used by application written using a different development tool. When your DLL contains VCL components (such as forms) that are to be utilized by the calling application, you need to provide exported interface routines that use standard calling conventions, avoid C++ name mangling, and do not require the calling application to support the VCL library in order to work.

For example, suppose we want to create a DLL to display the following simple dialog box:



The code for the dialog box DLL is as follows:

```
// DLLMAIN.H
//----------------------------------------------------------------------
#ifndef dllMainH
#define dllMainH
//----------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//----------------------------------------------------------------------
class TYesNoDialog : public TForm
{
__published:    // IDE-managed Components
    TLabel *LabelText;
    TButton *YesButton;
    TButton *NoButton;
    void __fastcall YesButtonClick(TObject *Sender);
    void __fastcall NoButtonClick(TObject *Sender);
private:        // User declarations
    bool returnValue;
public:         // User declarations
    virtual __fastcall TYesNoDialog(TComponent* Owner);
    bool __fastcall GetReturnValue();
};

// exported interface function
extern "C" __declspec(dllexport) bool InvokeYesNoDialog();

//----------------------------------------------------------------------
  extern TYesNoDialog *YesNoDialog;
//----------------------------------------------------------------------
#endif


// DLLMAIN.CPP
//----------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
```

```
#include "dllMain.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
__fastcall TYesNoDialog::TYesNoDialog(TComponent* Owner)
  : TForm(Owner)
{
   returnValue = false;
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::YesButtonClick(TObject *Sender)
{
    returnValue = true;
    Close();
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::NoButtonClick(TObject *Sender)
{
    returnValue = false;
    Close();
}
//---------------------------------------------------------------------------
bool __fastcall TYesNoDialog::GetReturnValue()
{
    return returnValue;
}
//---------------------------------------------------------------------------
// exported standard C++ interface function that calls into VCL
bool InvokeYesNoDialog()
{
    bool returnValue;
    TYesNoDialog *YesNoDialog = new TYesNoDialog(NULL);

    YesNoDialog->ShowModal();
    returnValue = YesNoDialog->GetReturnValue();
    delete YesNoDialog;

    return returnValue;
}
//---------------------------------------------------------------------------
```

The code in this example, displays the dialog and stores the value **true** in the private data member *returnValue* if the “Yes” button is pressed. Otherwise, *returnValue* is **false**. The public *GetReturnValue()* function retrieves the current value of *returnValue*.

To invoke the dialog and determine which button was pressed, the calling application calls the exported function *InvokeYesNoDialog()*. This function is declared in DLLMAIN.H as an exported function using C linkage (to avoid C++ name mangling) and the standard C calling convention. The function is defined in DLLMAIN.CPP.

By using a standard C function as the interface into the DLL, any calling application, whether or not it was created with C++Builder, can use the DLL. The VCL functionality required to support the dialog is linked into the DLL itself and the calling application does not need to know anything about it.

Note that when creating a DLL that uses the VCL, the required VCL components are linked into the DLL resulting in a certain amount of overhead. The impact of this overhead on the overall size of the application can be minimized by combining several components into one DLL which only needs one copy of the VCL support components.

DLLs and String objects

When a DLL includes exported functions that either take *String* (or *AnsiString*) objects as parameters or return *String* objects, even if those *String* objects are embedded in objects or structures, the following rules apply:

- The SHAREMEM.HPP header file must be included in both the source file(s) for the DLL that use these functions and the source file(s) for the .EXE that will call these functions.
- The file BCBMM.DLL (the Delphi shared-memory manager) must be deployed along with the DLL.

Use standard C++ strings instead of *String* objects if you do not want the DLL to require BCBMM.DLL.

ANSI Implementation-specific standards

Certain aspects of the ANSI C standard are not defined exactly by ANSI. Instead, each implementor of a C compiler is free to define these aspects individually. This topic describes how Borland has chosen to define these implementation-specific standards. The section numbers refer to the February 1990 ANSI Standard. Remember that there are differences between C and C++; this topic addresses C only.

2.1.1.3 How to identify a diagnostic.

When the compiler runs with the correct combination of options, any messages it issues beginning with the words *Fatal, Error,* or *Warning* are diagnostics in the sense that ANSI specifies. The options needed to ensure this interpretation are as follows:

Options needed for ANSI compliance

Option	**Action**
–A	Enable only ANSI keywords.
–C–	No nested comments allowed.
–i32	At least 32 significant characters in identifiers.
–p–	Use C calling conventions.
–w–	Turn off all warnings except the following.
–wbei	Turn on warning about inappropriate initializers.
–wbig	Turn on warning about constants being too large.
–wcpt	Turn on warning about nonportable pointer comparisons.
–wdcl	Turn on warning about declarations without type or storage class.
–wdup	Turn on warning about duplicate nonidentical macro definitions.
–wext	Turn on warning about variables declared both as external and as static.
–wfdt	Turn on warning about function definitions using a typedef.
–wrpt	Turn on warning about nonportable pointer conversion.
–wstu	Turn on warning about undefined structures.
–wsus	Turn on warning about suspicious pointer conversion.
–wucp	Turn on warning about mixing pointers to signed and unsigned char.
–wvrt	Turn on warning about void functions returning a value.
You cannot use the following options:	
–ms!	SS must be the same as DS for small data models.
–mm!	SS must be the same as DS for small data models.
–mt!	SS must be the same as DS for small data models.
–zG*xx*	The BSS group name cannot be changed.
–zS*xx*	The data group name cannot be changed.

Other options not specifically mentioned here can be set to whatever you want.

2.1.2.2.1 The semantics of the arguments to main.

The value of *argv*[0] is a pointer to a null byte when the program is run on DOS versions prior to version 3.0. For DOS version 3.0 or later, *argv*[0] points to the program name.

The remaining *argv* strings point to each component of the DOS command-line arguments. Whitespace separating arguments is removed, and each sequence of contiguous non-whitespace characters is treated as a single argument. Quoted strings are handled correctly (that is, as one string containing spaces).

2.1.2.3 What constitutes an interactive device.

An interactive device is any device that looks like the console.

2.2.1 The collation sequence of the execution character set.

The collation sequence for the execution character set uses the signed value of the character in ASCII.

2.2.1 Members of the source and execution character sets.

The source and execution character sets are the extended ASCII set supported by the IBM PC. Any character other than Ctrl+Z can appear in string literals, character constants, or comments.

2.2.1.2 Multibyte characters.

Multibyte characters are supported in C++Builder.

2.2.2 The direction of printing.

Printing is from left-to-right, the normal direction for the PC.

2.2.4.2 The number of bits in a character in the execution character set.

There are 8 bits per character in the execution character set.

3.1.2 The number of significant initial characters in identifiers.

The first 32 characters are significant, although you can use a command-line option (–**i**) to change that number. Both internal and external identifiers use the same number of significant characters. (The number of significant characters in C++ identifiers is unlimited.)

3.1.2 Whether case distinctions are significant in external identifiers.

The compiler normally forces the linker to distinguish between uppercase and lowercase. You can use a command-line option (–**l–c**) to suppress the distinction.

3.1.2.5 The representations and sets of values of the various types of integers.

Identifying diagnostics in C++

Type	16-bit minimum value	16-bit maximum value	32-bit minimum value	32-bit maximum value
signed char	–128	127	–128	127
unsigned char	0	255	0	255
signed short	–32,768	32,767	–32,768	32,767
unsigned short	0	65,535	0	65,535
signed int	–32,768	32,767	–2,147,483,648	–2,147,483,647
unsigned int	0	65,535	0	4,294,967,295
signed long	–2,147,483,648	2,147,483,647	–2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295	0	4,294,967,295

All **char** types use one 8-bit byte for storage.

All **short** types use 2 bytes, whether in a 16- or 32-bit program.

All **short** and **int** types use 2 bytes (in 16-bit programs).

All **int** types use 4 bytes (in 32-bit programs).

All **long** types use 4 bytes.

If alignment is requested (–**a**), all non**char** integer type objects will be aligned to even byte boundaries. If the requested alignment is **–a4**, the result is 4-byte alignment. Character types are never aligned.

3.1.2.5 The representations and sets of values of the various types of floating-point numbers.

The IEEE floating-point formats as used by the Intel 8087 are used for all C++Builder floating-point types. The **float** type uses 32-bit IEEE real format. The **double** type uses 64-bit IEEE real format. The **long double** type uses 80-bit IEEE extended real format.

3.1.3.4 The mapping between source and execution character sets.

Any characters in string literals or character constants remain unchanged in the executing program. The source and execution character sets are the same.

3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant.

Wide characters are supported.

3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.

Wide character constants are recognized.

3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character.

Character constants can contain one or two characters. If two characters are included, the first character occupies the low-order byte of the constant, and the second character occupies the high-order byte.

3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.

These conversions are performed by simply truncating the high-order bits. Signed integers are stored as two’s complement values, so the resulting number is interpreted as such a value. If the high-order bit of the smaller integer is nonzero, the value is interpreted as a negative value; otherwise, it is positive.

3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.

The integer value is rounded to the nearest representable value. Thus, for example, the **long** value (231 –1) is converted to the **float** value 231. Ties are broken according to the rules of IEEE standard arithmetic.

3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.

The value is rounded to the nearest representable value. Ties are broken according to the rules of IEEE standard arithmetic.

3.3 The results of bitwise operations on signed integers.

The bitwise operators apply to signed integers as if they were their corresponding unsigned types. The sign bit is treated as a normal data bit. The result is then interpreted as a normal two’s complement signed integer.

3.3.2.3 What happens when a member of a union object is accessed using a member of a different type.

The access is allowed and the different type member will access the bits stored there. You’ll need a detailed understanding of the bit encodings of floating-point values to understand how to access a floating-type member using a different member. If the member stored is shorter than the member used to access the value, the excess bits have the value they had before the short member was stored.

3.3.3.4 The type of integer required to hold the maximum size of an array.

For a normal array, the type is **unsigned int**, and for huge arrays the type is **signed long.**

3.3.4 The result of casting a pointer to an integer or vice versa.

When converting between integers and pointers of the same size, no bits are changed. When converting from a longer type to a shorter type, the high-order bits are truncated. When converting from a shorter integer type to a longer pointer type, the integer is first widened to an integer type the same size as the pointer type.

Thus signed integers will sign-extend to fill the new bytes. Similarly, smaller pointer types being converted to larger integer types will first be widened to a pointer type as wide as the integer type.

3.3.5 The sign of the remainder on integer division.

The sign of the remainder is negative when only one of the operands is negative. If neither or both operands are negative, the remainder is positive.

3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t.

The type is signed **int** when the pointers are **near** (or the program is a 32-bit application), or **signed long** when the pointers are **far** or **huge**. The type of ptrdiff_t depends on the memory model in use. In small data models, the type is **int**. In large data models, the type is **long**.

3.3.7 The result of a right shift of a negative signed integral type.

A negative signed value is sign extended when right shifted.

3.5.1 The extent to which objects can actually be placed in registers by using the *register* storage-class specifier.

Objects declared with any one, two, or four-byte integer or pointer types can be placed in registers. At least two and as many as seven registers are available. The number of registers actually used depends on what registers are needed for temporary values in the function.

3.5.2.1 Whether a plain int bit-field is treated as a signed int or as an unsigned int bit field.

Plain **int** bit fields are treated as **signed int** bit fields.

3.5.2.1 The order of allocation of bit fields within an int.

Bit fields are allocated from the low-order bit position to the high-order.

3.5.2.1 The padding and alignment of members of structures.

By default, no padding is used in structures. If you use the word alignment option (–**a**), structures are padded to even size, and any members that do not have character or character array type are aligned to an even multiple offset.

3.5.2.1 Whether a bit-field can straddle a storage-unit boundary.

When alignment (–**a**) is not requested, bit fields can straddle dword boundaries, but are never stored in more than four adjacent bytes.

3.5.2.2 The integer type chosen to represent the values of an enumeration type.

Store all enumerators as full **int**s. Store the enumerations in a **long** or **unsigned long** if the values don't fit into an **int**. This is the default behavior as specified by **–b** compiler option.

The **–b**- behavior specifies that enumerations should be stored in the smallest integer type that can represent the values. This includes all integral types, for example, **signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long,** and **unsigned long**.

For C++ compliance, **–b-** must be specified because it is not correct to store all enumerations as **int**s for C++.

3.5.3 What constitutes an access to an object that has volatile-qualified type.

Any reference to a volatile object will access the object. Whether accessing adjacent memory locations will also access an object depends on how the memory is constructed in the hardware. For special device memory, such as video display memory, it depends on how the device is constructed. For normal PC memory, volatile objects are used only for memory that might be accessed by asynchronous interrupts, so accessing adjacent objects has no effect.

3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type.

There is no specific limit on the number of declarators. The number of declarators allowed is fairly large, but when nested deeply within a set of blocks in a function, the number of declarators will be reduced. The number allowed at file level is at least 50.

3.6.4.2 The maximum number of case values in a switch statement.

There is no specific limit on the number of cases in a switch. As long as there is enough memory to hold the case information, the compiler will accept them.

3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value.

All character constants, even constants in conditional directives, use the same character set (execution). Single-character character constants will be negative if the character type is signed (default and –**K** not requested).

3.8.2 The method for locating includable source files.

For include file names given with angle brackets, if include directories are given in the command line, then the file is searched for in each of the include directories. Include directories are searched in this order: first, using directories specified on the command line, then using directories specified in TURBOC.CFG or BCC32.CFG. If no include directories are specified, then only the current directory is searched.

3.8.2 The support for quoted names for includable source files.

For quoted file names, the file is first searched for in the current directory. If not found, searches for the file as if it were in angle brackets.

3.8.2 The mapping of source file name character sequences.

Backslashes in include file names are treated as distinct characters, not as escape characters. Case differences are ignored for letters.

3.8.8 The definitions for _ _DATE_ _ and _ _TIME_ _ when they are unavailable.

The date and time are always available and will use the operating system date and time.

4.1.1 The decimal point character.

The decimal point character is a period (.).

4.1.5 The type of the sizeof operator, *size_t*.

The type *size_t is* ***unsigned.***

4.1.5 The null pointer constant to which the macro NULL expands.

For a 16-bit application, an integer or a long 0, depending on the memory model.

For 32-bit applications, NULL expands to an **int** zero or a **long** zero. Both are 32-bit **signed** numbers.

4.2 The diagnostic printed by and the termination behavior of the assert function.

The diagnostic message printed is “Assertion failed: *expression*, file *filename*, line *nn*”, where *expression* is the asserted expression that failed, *filename* is the source file name, and *nn* is the line number where the assertion took place.

Abort is called immediately after the assertion message is displayed.

4.3 The implementation-defined aspects of character testing and case-mapping functions.
None, other than what is mentioned in 4.3.1.

4.3.1 The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint and isupper functions.
First 128 ASCII characters for the default C locale. Otherwise, all 256 characters.

4.5.1 The values returned by the mathematics functions on domain errors.
An IEEE NAN (not a number).

4.5.1 Whether the mathematics functions set the integer expression *errno* to the value of the macro ERANGE on underflow range errors.
No, only for the other errors—domain, singularity, overflow, and total loss of precision.

4.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero.
No; fmod(x,0) returns 0.

4.7.1.1 The set of signals for the signal function.
SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM.

4.7.1.1 The semantics for each signal recognized by the signal function.
See the description of signal.

4.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function.
See the description of signal.

4.7.1.1 If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed.
The equivalent of signal(sig, SIG_DFL) is always executed.

4.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function.
No, it is not.

4.9.2 Whether the last line of a text stream requires a terminating newline character.
No, none is required.

4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in.
Yes, they do.

4.9.2 The number of null characters that may be appended to data written to a binary stream.
None.

4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.
The file position indicator of an append-mode stream is initially placed at the beginning of the file. It is reset to the end of the file before each write.

4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point.
A write of 0 bytes might or might not truncate the file, depending on how the file is buffered. It is safest to classify a zero-length write as having indeterminate behavior.

4.9.3 The characteristics of file buffering.

Files can be fully buffered, line buffered, or unbuffered. If a file is buffered, a default buffer of 512 bytes is created upon opening the file.

4.9.3 Whether a zero-length file actually exists.

Yes, it does.

4.9.3 Whether the same file can be open multiple times.

Yes, it can.

4.9.4.1 The effect of the remove function on an open file.

No special checking for an already open file is performed; the responsibility is left up to the programmer.

4.9.4.2 The effect if a file with the new name exists prior to a call to rename.

Rename returns a –1 and *errno* is set to EEXIST.

4.9.6.1 The output for %p conversion in fprintf.

In near data models, four hex digits (XXXX). In far data models, four hex digits, colon, four hex digits (XXXX:XXXX). (For 16-bit programs.)

Eight hex digits (XXXXXXXX). (For 32-bit programs.)

4.9.6.2 The input for %p conversion in fscanf.

See 4.9.6.1.

4.9.6.2 The interpretation of a –(hyphen) character that is neither the first nor the last character in the scanlist for a %[conversion in fscanf.

See the description of scanf**.**

4.9.9.1 The value the macro errno is set to by the fgetpos or ftell function on failure.

EBADF Bad file number.

4.9.10.4 The messages generated by perror.

Messages generated in both Win16 and Win32

Arg list too big	Math argument
Attempted to remove current directory	Memory arena trashed
Bad address	Name too long
Bad file number	No child processes
Block device required	No more files
Broken pipe	No space left on device
Cross-device link	No such device
Error 0	No such device or address
Exec format error	No such file or directory
Executable file in use	No such process
File already exists	Not a directory
File too large	Not enough memory
Illegal seek	Not same device
Inappropriate I/O control operation	Operation not permitted
Input/output error	Path not found
Interrupted function call	Permission denied

Invalid access code	Possible deadlock
Invalid argument	Read-only file system
Invalid data	Resource busy
Invalid environment	Resource temporarily unavailable
Invalid format	Result too large
Invalid function number	Too many links
Invalid memory block address	Too many open files
Is a directory	

Messages generated only in Win32

Bad address	No child processes
Block device required	No space left on device
Broken pipe	No such device or address
Executable file in use	No such process
File too large	Not a directory
Illegal seek	Operation not permitted
Inappropriate I/O control operation	Possible deadlock
Input/output error	Read-only file system
Interrupted function call	Resource busy
Is a directory	Resource temporarily unavailable
Name too long	Too many links

4.10.3 The behavior of calloc, malloc, or realloc if the size requested is zero.

calloc and malloc will ignore the request and return 0. realloc will free the block.

4.10.4.1 The behavior of the abort function with regard to open and temporary files.

The file buffers are not flushed and the files are not closed.

4.10.4.3 The status returned by exit if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE.

Nothing special. The status is returned exactly as it is passed. The status is a represented as a **signed char.**

4.10.4.4 The set of environment names and the method for altering the environment list used by getenv.

The environment strings are those defined in the operating system with the SET command. *putenv* can be used to change the strings for the duration of the current program, but the SET command must be used to change an environment string permanently.

4.10.4.5 The contents and mode of execution of the string by the system function.

The string is interpreted as an operating system command. COMSPEC is used or COMMAND.COM is executed (for 16-bit programs) or CMD.EXE (for 32-bit programs) and the argument string is passed as a command to execute. Any operating system built-in command, as well as batch files and executable programs, can be executed.

4.11.6.2 The contents of the error message strings returned by strerror.

See 4.9.10.4.

4.12.1 The local time zone and Daylight Saving Time.

Defined as local PC time and date.

4.12.2.1 The era for clock.

Represented as clock ticks, with the origin being the beginning of the program execution.

4.12.3.5 The formats for date and time.

C++Builder implements ANSI formats.

Keywords

{button A,JI(`',`keywords_a')} {button B,JI(`',`keywords_b')} {button C,JI(`',`keywords_c')} {button D,JI(`',`keywords_d')} {button E,JI(`',`keywords_e')} {button F,JI(`',`keywords_f')} {button G,JI(`',`keywords_g')} {button H,JI(`',`keywords_h')} {button I,JI(`',`keywords_i')} {button J,JI(`',`keywords_i')} {button l,JI(`',`keywords_l')} {button N,JI(`',`keywords_n')} {button O,JI(`',`keywords_o')} {button P,JI(`',`keywords_p')} {button R,JI(`',`keywords_r')} {button S,JI(`',`keywords_s')} {button T,JI(`',`keywords_t')} {button U,JI(`',`keywords_u')} {button V,JI(`',`keywords_v')} {button W,JI(`',`keywords_w')}

Keywords are words reserved for special purposes and must **not** be used as normal identifier names

This is an alphabetical listing of the keywords supported in this release of C++Builder. The description of some keywords is provided only under the topic to which they apply. For example, C++ cast operators are discussed under New-style typecasting, and the discussion of keywords related to C++ namespaces is discussed under the Namespaces overview.

For a functional listing of the keywords, see Keywords (by Category).

A

__asm

_asm

asm

auto

__automated

B

break

bool

C

case

catch

__cdecl

_cdecl

cdecl

char

class

__classid

__closure

const

const_cast

continue

D

__declspec

default

delete

__dispid

do

double

dynamic_cast

E

else

enum

__except
explicit
_export
__export
extern

F
false
__fastcall
_fastcall
__finally
float
for
friend

G
goto

I
if
__import
_import
inline
int
__int8
__int16
__int32
__int64

L
long
mutable

N
namespace
new

O
operator

P
__pascal
_pascal
pascal
private
protected
__property
public
__published

R

register

reinterpret_cast

return

__rtti

S

short

signed

sizeof

static

static_cast

__stdcall

_stdcall

struct

switch

T

template

this

__thread

throw

true

__try

try

typedef

typename

typeid

U

union

using

unsigned

V

virtual

void

volatile

W

wchar_t

while

Keywords (by Category)

This is a categorical listing of the keywords C++Builder supports. For an alphabetical listing of the keywords, see Keywords (Alphabetical).

C++Builder Extensions	keywords unique to C++Builder
C++ Specific	keywords recognized only in C++ programs
Modifiers	keywords that change one or more attributes of an identifier associated with an object
Operators	keywords that invoke functions against objects or identifiers
Statements	keywords that specify program control during execution
Storage Class Specifiers	keywords that define the location and duration of an identifier
Type Specifiers	keywords that determine how memory is allocated and bit patterns are interpreted

Modifiers

A declaration uses modifiers to alter aspects of the identifier/object mapping.

The C++Builder modifiers are:

__cdecl
const
__declspec
__dispid
__export
__fastcall
__import
__pascal
__rtti
__stdcall
volatile

Operator Keywords

See also

Several C++Builder keywords denote operators that invoke functions against objects and identifiers.

The keyword operators supported by C++Builder are:

delete

operator

typeid

new

sizeof

__classid

Statement Keywords

Statements specify the flow of control in a program. In the absence of specific jumps and selection statements, statements execute sequentially as they appear in the source code.

The statement keywords in C++Builder are:

break	else	switch
case	__finally	throw
catch	for	__try
continue	goto	try
default	if	while
do	return	
__except		

asm, _asm, __asm

See also Example Keywords

Syntax

```
asm <opcode> <operands> <; or newline>
_asm <opcode> <operands> <; or newline>
__asm <opcode> <operands> <; or newline>
```

Description

Use the **asm, _asm**, or **__asm** keyword to place assembly language statements in the middle of your C or C++ source code. Any C++ symbols are replaced by the appropriate assembly language equivalents.

You can group assembly language statements by beginning the block of statements with the **asm** keyword, then surrounding the statements with braces (`{}`).

Examples

```
// This example places a single assembler statement in your code:
asm pop dx

// If you want to include several of asm statements,
// surround them with braces:
asm {
  mov ax, 0x0e07
  xor bx, bx
  int 0x10         // makes the system beep
}
```

auto

Example Keywords

Syntax

```
[auto] <data-definition> ;
```

Description

Use the **auto** modifer to define a local variable as having a local lifetime.

This is the default for local variables and is rarely used.

Example

```
int main()
{
  auto int i;
  i = 5;
  return i;
}
```

__automated

See also Keywords Example

Syntax

```
_automated: <declarations>
```

Description

The visibility rules for automated members are identical to those of public members. The only difference between automated and public members is that OLE automation information is generated for member functions and properties that are declared in an automated section. This OLE automation type information makes it possible to create OLE Automation servers.

- For a member function, the types of all member function parameters and the function result (if any) must be automatable. Likewise, for a property, the property type and the types of any array property parameters must be automatable. The automatable types are: *Currency*, **double**, **int**, **float**, **short**, *String*, *TDateTime*, *Variant*, and **unsigned short**. Declaring member functions or properties that use non-automatable types in an **__automated** section results in a compile-time error.
- Member function declarations must use the **__fastcall** calling convention.
- Member functions can be **virtual**.
- Member functions may add **__dispid**(*constant int expression*) after the closing parenthesis of the parameter list.
- Property declarations can only include access specifiers (**__dispid**, **read,** and **write**). No other specifiers (**index**, **stored**, **default**, **nodefault**) are allowed.
- Property access specifiers must list a member function identifier. Data member identifiers are not allowed.
- Property access member functions must use the **__fastcall** calling convention.
- Property overrides (property declarations that don’t include the property type) are not allowed.

Example of __automated and __dispid

```
// Illustrates the use of __automated and __dispid.
class myclass : TAutoObject
{
// This access region is used to declare functions
// and properties that need OLE automation information.
__automated :
   // The __dispid directive associates the OLE automation
   // dispatch id 1000 with the member function.
   void __fastcall func(void)  __dispid(1000);
};
```

bool

See also Example Keywords

Syntax

```
bool <identifier>;
```

Description

Use **bool** and the literals **false** and **true** to perform Boolean logic tests.

The **bool** keyword represents a type that can take only the value **false** or **true**. The keywords **false** and **true** are Boolean literals with predefined values. **false** is numericallly zero and **true** is numerically one. These Boolean literals are rvalues; you cannot make an assignment to them.

You can convert an rvalue that is of type **bool** to an rvalue that is **int** type. The numerical conversion sets **false** to zero and **true** becomes one.

You can convert arithmetic, enumeration, pointer, or pointer to member rvalue types to an rvalue of type **bool**. A zero value, null pointer value, or null member pointer value is converted to **false**. Any other value is converted to **true**.

Example

```
/* How to make Boolean tests with bool, true, and false. */
#include <iostream.h>

bool func() {     // Function returns a bool type
   return NULL;  // NULL is converted to Boolean false
// return false;  // This statement is Boolean equivalent to the one above.
   }

int main() {
   bool val = false;  // Boolean variable
   int i = 1;         // i is neither Boolean-true nor Boolean-false
   int *iptr = 0;     // null pointer
   float j = 1.01;    // j is neither Boolean-true nor Boolean-false

   // Tests on integers
   if (i == true)  cout << "True: value is 1" << endl;
   if (i == false) cout << "False: value is 0" << endl;

 // Test on pointer
   if (iptr == false) cout << "Invalid pointer." << endl;

   // To test j's truth value, cast it to bool type.
   if (bool(j) == true) cout << "Boolean j is true." << endl;

   // Test Boolean function return value
   val = func();
   if (val == false)
      cout << "func() returned false.";
   if (val == true)
      cout << "func() returned true.";
   return false;  // false is converted to 0
}
```

Program output

```
True: value is 1
Unknown truth value for g.
Invalid pointer.
Boolean j is true.
func() returned false.
```

break

See also Example Keywords

Syntax

```
break ;
```

Description

Use the **break** statement within loops to pass control to the first statement following the innermost **switch**, **for**, **while**, or **do** block.

Example

```c
/* Illustrates the use of keywords break, case, default, and switch. */
#include <conio.h>
#include <stdio.h>

int main(void) {
   int ch;

   printf("\tPRESS a, b, OR c. ANY OTHER CHOICE WILL "
          "TERMINATE THIS PROGRAM.");
   for ( /* FOREVER */; ((ch = getch()) != EOF); )
      switch (ch) {
         case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
            printf("\nOption a was selected.\n");
            break;
         case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
         case 'c' :
            printf("\nOption b or c was selected.\n");
            break;
         default :
            printf("\nNOT A VALID CHOICE!  Bye ...");
            return(-1);
         }
   return(0);
   }
```

case

See also Example Keywords

Syntax

```
switch ( <switch variable> ){
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;
}
```

Description

Use the **case** statement in conjunction with switches to determine which statements evalute.

The list of possible branch points within `<statement>` is determined by preceding substatements with

`case <constant expression> : <statement>;`

where `<constant expression>` must be an **int** and must be unique.

The `<constant expression>` values are searched for a match for the `<switch variable>`.

If a match is found, execution continues after the matching **case** statement until a **break** statement is encountered or the end of the **switch** statement is reached.

If no match is found, control is passed to the **default** case.

Note: It is illegal to have duplicate **case** constants in the same **switch** statement.

catch

See also Keywords

Syntax

```
catch (exception-declaration) compound-statement
```

Description

The exception handler is indicated by the **catch** keyword. The handler must be used immediately after the statements marked by the try keyword. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

cdecl, _cdecl, _ _cdecl

Example Keywords

Syntax

```
cdecl <data/function definition> ;
_cdecl <data/function definition> ;
__cdecl <data/function definition> ;
```

Description

Use a **cdecl**, **_cdecl**, or **_ _cdecl** modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use **cdecl**, **_cdecl**, or **_ _cdecl** in front of a function, it effects how the parameters are passed (last parameter is pushed first, and the caller cleans up the stack). The **_ _cdecl** modifier overrides the compiler directives and IDE options.

The **cdecl**, **_cdecl**, and **__cdecl** keywords are specific to C++Builder.

Example

```
int __cdecl FileCount;
long __cdecl HisFunc(int x);
```

char

See also Keywords

Syntax

```
[signed|unsigned] char <variable_name>
```

Description

Use the type specifier **char** to define a character data type. Variables of type **char** are 1 byte in length.

A **char** can be signed, unsigned, or unspecified. By default, **signed char** is assumed.

Objects declared as characters (**char**) are large enough to store any member of the basic ASCII character set.

class

See also Example Keywords

Syntax

`<classkey> <classname> <baselist> { <member list> }`

- `<classkey>` is either a class, struct, or union.
- `<classname>` can be any name unique within its scope.
- `<baselist>` lists the base class(es) that this class derives from. `<baselist>` is optional
- `<member list>` declares the class's data members and member functions.

Description

Use the **class** keyword to define a C++ class.

Within a class:

- the data are called data members
- the functions are called member functions

Example

```
class stars {
   int magnitude;        // Data member
   int starfunc(void);   // Member function
};
```

__classid

See also Keywords Example

Syntax

```
__classid(classType)
```

Description

The **__classid** operator returns a pointer to the vtable for the specified *classType*. The operator is used internally by the VCL support classes and by the generated project source code to interact with VCL methods (also known as *member functions*) that use class parameters.

This operator should not be directly used by C++Builder programmers.

__closure

See also Example Keywords

Syntax

```
<type> ( __closure * <id> ) (<param list>);
```

Description

Use **__closure** to declare event handler functions.

A closure declaration is the same as a function pointer declaration but with the addition of the **__closure** keyword in front of the *<id>* being defined. While a function pointer contains only a 4-byte code address, a closure contains both a code address (in the first 4 byte) and an object pointer (in the second 4 bytes) that serves as the **this** pointer when you call through the closure.

closure example

```
struct MyObject
{
   double MemFunc(int);
};
double func1(MyObject *o)
{
   // A pointer to a member function taking an
   // int argument and returning double.
   double ( __closure *myClosure )(int);

   // Initialize the closure.
   myClosure = o -> MemFunc;

   // Use the closure to call the member function and pass it an int.
   return myClosure(1);
}
```

const

See also Example Keywords

Syntax

```
const <variable name> [ = <value> ] ;
<function name> ( const <type>*<variable name> ;)
<function name> const;
```

Description

Use the **const** modifier to make a variable value unmodifiable.

Use the **const** modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a **const** result in a compiler error.

A **const** pointer cannot be modified, though the object to which it points can be changed. Consider the following examples.

```
   const float pi   = 3.14;
   const  maxint  = 12345;   // When used by itself, const is equivalent to
  int.
   char *const str1 = "Hello, world";           // A constant pointer
   char const *str2 = "Borland International";  // A pointer to a constant
  character string.
```

Given these declarations, the following statements are illegal.

```
pi   = 3.0;          // Assigns a value to a const.
i     = maxint++;    // Increments a const.
str1 = "Hi, there!" // Points str1 to something else.
```

Using the const Keyword in C++ Programs

C++ extends **const** to include classes and member functions. In a C++ class definition, use the **const** modifier following a member function declaration. The member function is prevented from modifying any data in the class.

A class object defined with the **const** keyword attempts to use only member functions that are also defined with **const**. If you call a member function that is not defined as **const**, the compiler issues a warning that the a non-const function is being called for a **const** object. Using the **const** keyword in this manner is a safety feature of C++.

Warning: A pointer can indirectly modify a **const** variable, as in the following:

*(int *)&my_age = 35;

If you use the **const** modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int printf (const char *format, ...);
```

printf is prevented from modifying the format string.

Example

```
class X  {
   int j;
public:
  X::X() { j = 0; };
  int lowerBound() const;             // DOES NOT MODIFY ANY DATA MEMBERS
  int dimension(X x1, const X &x2) { // x2 PARAMETERS WON'T BE MODIFIED
     x1.j = 3;       // OKAY; x1 OBJECT IS MODIFIABLE
     x2.j = 5;       // ERROR; x2 IS NOT MODIFIABLE
     return x2.j;
     }
};
```

Example 2

```
#include <iostream.h>

class Alpha {
   int num;
public:
   Alpha(int j = 0) { num = j; }
   int func(int i) const {
      cout << "Non-modifying function." << endl;
      return i++;
      }
   int func(int i) {
      cout << "Modify private data" << endl;
      return num = i;
      }
   int f(int i) { cout << "Non-const function called with i = " << i <<
  endl; return i;}
};

void main() {
   Alpha alpha_mod;          // Calls the non-const functions.
   const Alpha alpha_inst;  // Attempts to call the const functions.

   alpha_mod.func(1);
   alpha_mod.f(1);           // Causes a compiler warning.

   alpha_inst.func(1);
   alpha_inst.f(1);
   }
```

Output:

```
Modify private data
Non-const function called with i = 1
Non-modifying function.
Non-const function called with i = 1
```

continue

See also Example Keywords

Syntax

```
continue ;
```

Description

Use the **continue** statement within loops to pass control to the end of the innermost enclosing brace; at which point the loop continuation condition is re-evaluated.

Example

```
void main ()
{
  for (i = 0; i < 20; i++) {
     if (array[i] == 0)
       continue;
     array[i] = 1/array[i];
  }
}
```

_ _declspec

See also Example Keywords

Syntax

```
__declspec(<decl-modifier>)
```

Description

Use the **_ _declspec** keyword to indicate the storage class attributes for a variable or function.

The **_ _declspec** keyword is required when making forward declarations of VCL classes.

The **_ _declspec** keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The **_ _declspec** keyword and its argument can appear anywhere in the declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);                       // illegal
void __export f(void)                        // correct
void __declspec(dllexport) f(void);          // correct
__declspec(dllexport)void f(void);           // correct
class __declspec(dllexport) ClassName { }    // correct
```

The *decl-modifier* argument can only be one of *dllexport*, *dllimport*, or *thread*. The meaning of these arguments is equivalent to the following storage class attribute keywords.

Argument	Storage Class	Compiler Support
dllexport	__export	32- and 16-bit
dllimport	__import	32-bit (legal, but no affect on 16-bit programs)
thread	__thread	32-bit only

Example

```
/* Examples of  __declspec declarations follow. */
__declspec(dllimport) void func(void);
__declspec(dllimport) int a;
__declspec(dllexport) void bar (void);

/** Use thread argument only with static storage data. **/
__declspec(thread) int th;
int __declspec(thread) th1;
```

Using __declspec in VCL class declarations

```
// Forward declarations
class __declspec(delphiclass) MyVclClass;
class __declspec(delphireturn) MyString;
MyVclClass* MyVclClassFunc();
MyString* MyStringFunc();

class MyClass : public TVclClass
{   /* class definition */ };

class MyString: public AnsiString
{  /* class definition */ };
```

default

See also Example Keywords

Syntax

```
switch ( <switch variable> ){
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;
}
```

Description

Use the default statement in switch statement blocks.

- If a case match is not found and the **default** statement is found within the switch statement, the execution continues at this point.
- If no default is defined in the **switch** statement, control passes to the next statement that follows the switch statement block.

__dispid

See also Keywords Example

Syntax

`__dispid(`*`constant int expression`*`)`

Description

A member function that has been declared in the **__automated** section of a class can include an optional **__dispid**(*constant int expression)* directive. The directive must be declared after the closing parenthesis of the parameter list.

The *constant int expression* gives the OLE Automation dispatch ID of the member function or property. If a **dispid** directive is not used, the compiler automatically picks a number one larger than the largest dispatch ID used by any member function or property in the class and its base classes.

Specifying an already-used dispatch ID in a dispid directive causes a compile-time error.

do

See also Example Keywords

Syntax

```
do <statement> while ( <condition> );
```

Description

The **do** statement executes until the condition becomes **false**.

`<statement>` is executed repeatedly as long as the value of `<condition>` remains **true**.

Since the conditon tests after each the loop executes the `<statement>`, the loop will execute at least once.

do Example

```
/* This example prompts users for a password  */
/* and continued to prompt them until they    */
/* enter one that matches the value stored in */
/* checkword.                                 */

#include <stdio.h>
#include <string.h>

int main ()
{
  char checkword[80] = "password";
  char password[80]  = "";

  do {
    printf ("Enter password: ");
    scanf("%s", password);
  } while (strcmp(password, checkword));

  return 0;
}
```

double

See also Keywords

Syntax

```
[long] double <identifier>
```

Description

Use the **double** type specifier to define an identifier to be a floating-point data type. The optional modifier **long** extends the accuracy of the floating-point value.

If you use the **double** keyword, the C++Builder IDE will automatically link the floating-point math package into your program.

enum

See also Example Keywords

Syntax

```
enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];
```

- `<type_tag>` is an optional type tag that names the set.
- `<constant_name>` is the name of a constant that can optionally be assigned the value of `<value>`. These are also called enumeration constants.
- `<value>` must be an integer. If `<value>` is missing, it is assumed to be:

  `<prev> + 1`

  where `<prev>` is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0.
- `<var_list>` is an optional variable list that assigns variables to the enum type.

Description

Use the **enum** keyword to define a set of constants of type **int**, called an enumeration data type.

An enumeration data type provides mnemonic identifiers for a set of integer values. C++Builder stores enumerators in a single byte if you uncheck Treat Enums As Ints (O|C|Code Generation) or use the **-b** flag.

Enums are always interpreted as **int**s if the range of values permits, but if they are not **int**s the value gets promoted to an **int** in expressions. Depending on the values of the enumerators, identifiers in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**.

In C, an enumerated variable can be assigned any value of type int--no type checking beyond that is enforced. In C++, an enumerated variable can be assigned only one of its enumerators.

In C++, lets you omit the **enum** keyword if `<tag_type>` is not the name of anything else in the same scope. You can also omit `<tag_type>` if no further variables of this **enum** type are required.

In the absence of a `<value>` the first enumerator is assigned the value of zero. Any subsequent names without initializers will then increase by one. `<value>` can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers.

In C++, enumerators declared within a class are in the scope of that class.

Examples

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, enum days, a variable anyday of this type, and a set of enumerators (sun, mon,...) with constant integer values.

```
enum modes { LASTMODE = -1, BW40=0, C40, BW80, C80, MONO = 7 };
/*
   "modes" is the type tag.
   "LASTMODE", "BW40", "C40", etc. are the constant names.
   The value of C40 is 1 (BW40 + 1); BW80 = 2 (C40 + 1), etc.
*/
```

_ _except

See also Keywords

Syntax

```
__except (expression) compound-statement
```

Description

The **_ _except** keyword specifies the action that should be taken when the exception specified by *expression* has been raised.

explicit

See also

Syntax

```
explicit <single-parameter constructor declaration>
```

Description

Normally, a class with a single-parameter constructor can be assigned a value that matches the constructor type. This value is automatically (implicitly) converted into an object of the class type to which it is being assigned. You can prevent this kind of implicit conversion from occurring by declaring the constructor of the class with the **explicit** keyword. Then all objects of that class must be assigned values that are of the class type; all other assignments result in a compiler error.

Objects of the following class can be assigned values that match the constructor type or the class type:

```
class X {
public:
  X(int);
  X(const char*, int = 0);
};
```

Then, the following assignment statements are legal.

```
void f(X arg) {
  X a = 1;
  X B = "Jessie";
  a = 2;
}
```

However, objects of the following class can be assigned values that match the class type only:

```
class X {
public:
  explicit X(int);
  explicit X(const char*, int = 0);
};
```

The **explicit** constructors then require the values in the following assignment statements to be converted to the class type to which they are being assigned.

```
void f(X arg) {
  X a = X(1);
  X b = X("Jessie",0);
  a = X(2);
}
```

_export, _ _export

See also Keywords

Form 1

```
class _export <class name>
```

Form 2

```
return_type _export <function name>
```

Form 3

```
data_type _export <data name>
```

Description

These modifiers are used to export classes, functions, and data.

The linker enters functions flagged with **_export** or **_ _export** into an export table for the module.

Using **_export** or **_ _export** eliminates the need for an EXPORTS section in your module definition file.

Functions that are not modified with **_export** or **_ _export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note: If you use **_export** or **_ _export** to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need a module definition file.

Compiler Options and the _ _export Keyword

See also Keywords

This table summarizes the effect of the combination of various Windows options and the **_ _export** keyword:

The compiler option is: *	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**	**-tW**·or **-tWD**	**-tWE**·or **-tWDE**
Function flagged with __export?	Yes	Yes	Yes	Yes	No	No	No	No
Function·listed in EXPORTS?	Yes	Yes	No	No	Yes	Yes	No	No
Is·function exportable?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Will function be exported?	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**No**	**No**

* Or the 32-bit console-mode application equivalents.

Smart Callbacks and the _export Keyword

See also Keywords

If you use the Smart Callbacks IDE option at compile time, callback functions do not need to be listed in the EXPORTS statement or flagged with the **_export** keyword.

Functions compile them so that they are callback functions.

Exportable Functions in DLLs

See also

There are two ways to compile a function `f1( )` in a DLL as exportable and then export it.

- Compile the DLL with all functions exportable (with the Windows DLL All Functions Exportable option in the IDE) and list `f1( )` in the EXPORTS section of the module definition file, or
- Flag the function `f1( )` with the _export keyword.

Using _export with C++ Classes

See also Keywords

Whenever you declare a class as **_export**, the compiler exports all of its non-inline member functions and static data members.

If you declare the class in an include file that is included in both the DLL source files and the source files of the application that use the DLL, declare the class as **_export** when compiling the DLL

To do this, use the __DLL__ macro, which the compiler defines when it's building a DLL.

extern

See also Example Keywords

Syntax

```
extern <data definition> ;
[extern] <function prototype> ;
```

Description

Use the **extern** modifier to indicate that the actual storage and initial value of a variable, or body of a function, is defined in a separate source code module. Functions declared with extern are visible throughout all source files in a program, unless you redefine the function as static.

The keyword **extern** is optional for a function prototype.

Use `extern "c"` to prevent function names from being mangled in C++ programs.

Examples

```
extern int _fmode;
extern void Factorial(int n);
extern "c" void cfunc(int);
```

Parameter Types and Possible Registers Used

The compiler uses the following rules when deciding which parameters are to be passed in registers.

Parameter Type	Registers
char (signed and unsigned)	AL, DL, BL
int (signed and unsigned)	AX, DX, BX
long (signed and unsigned)	DX:AX
near pointer	AX, DX, BX

Only three parameters can be passed in registers to any one function.

Do not assume the assignment of registers will reflect the ordering of the parameters to a function. Union, structure, and floating-point (**float**, **double**, and **long**) parameters are pushed on the stack.

_fastcall, _ _fastcall

See also Keywords

Syntax

```
return-value _fastcall function-name(parm-list)
return-value __fastcall function-name(parm-list)
```

Description

Use the **__fastcall** modifier to declare functions that expect parameters to be passed in registers. The first three parameters are passed (from left to right) in EAX, EBX, and EDX, if they fit in the register. The registers are not used if the parameter is a floating-point or struct type.

All form class member functions must use the **__fastcall** convention.

The compiler treats this calling convention as a new language specifier, along the lines of _cdecl and _pascal

Functions declared using **_cdecl** or **_pascal** cannot also have the **_fastcall** modifiers because they use the stack to pass parameters. Likewise, the **__fastcall** modifier cannot be used together with _export.

The compiler prefixes the **_ _fastcall** function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

Note: The **__fastcall** modifier is subject to name mangling.

_ _finally

See also Keywords

Syntax

```
__finally {compound-statement}
```

Description

The **_ _finally** keyword specifies actions that should be taken regardless of how the flow within the preceding _ _try exits.

The **_ _finally** keyword is supported only in C programs.

float

See also Keywords

Syntax

```
float <identifier>
```

Description

Use the **float** type specifier to define an identifier to be a floating-point data type.

Type	Length	Range
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)

The C++Builder IDE automatically links the floating-point math package into you program if you use floating-point values or operators.

for

Example Keywords

Syntax

```
for ( [<initialization>] ; [<condition>] ; [<increment>] )  <statement>
```

Description

The **for** statement implements an iterative loop.

`<condition>` is checked before the first entry into the block.

`<statement>` is executed repeatedly UNTIL the value of `<condition>` is false.

- Before the first iteration of the loop, `<initialization>` initializes variables for the loop.
- After each iteration of the loop, `<increments>` increments a loop counter. Consequently, `j++` is functionally the same as `++j`.

In C++, `<initialization>` can be an expression or a declaration.

The scope of any identifier declared within the **for** loop extends to the end of the control statement only.

A variable defined in the **for**-*initialization* expression is in scope only within the **for**-block. See the description of the -Vd option.

All the expressions are optional. If `<condition>` is left out, it is assumed to be always true.

Examples

```
// An example of the scope of variables in for-expressions.
// The example compiles if you use the -Vd option.
#include <iostream.h>

int main() {
   for (int i = 0; i < 10; i++)
      if (i == 8)
      cout << "\ni = " << i;
return i;  // Undefined symbol ‘i’ in function main().
}
```

friend

Example Keywords

Syntax

```
friend <identifier>;
```

Description

Use **friend** to declare a function or class with full access rights to the private and protected members of an outside class, without being a member of that class.

In all other respects, the **friend** is a normal function in terms of scope, declarations, and definitions.

Example

```
class stars {
   friend class galaxy;
   int magnitude;
   int starfunc(void);
};

class galaxy {
   long int number_of_stars;
   void stars_magnitude(stars&);
   void stars_func(stars*);
}
```

goto

Example Keywords

Syntax

```
goto <identifier> ;
```

Description

Use the **goto** statement to transfer control to the location of a local label specified by <identifier>.

Labels are always terminated by a colon.

Example

```
Again:       /* this is the label */
;
.
.
.
goto Again;
```

if

See also Example Keywords

Syntax

```
if ( <condition> )  <statement1>;

if ( <condition> )  <statement1>;
else  <statement2>;
```

Description

Use **if** to implement a conditional statement.

You can declare variables in the condition expression. For example,

```
   if (int val = func(arg))
```

is valid syntax. The variable *val* is in scope for the **if** statement and extends to an **else** block when it exists.

The condition statement must convert to a **bool** type. Otherwise, the condition is ill-formed.

When `<condition>` evaluates to **true**, `<statement1>` executes.

If `<condition>` is **false**, `<statement2>` executes.

The **else** keyword is optional, but no statements can come between an **if** statement and an **else**.

The #if and #else preprocessor statements (directives) look similar to the **if** and **else** statements, but have very different effects. They control which source file lines are compiled and which are ignored.

Examples

```
if (int val = func(count)) { /* statements */ }
else {
   /* take other action */
   cout << “val is false”
   }
```

_import, _ _import

Keywords

Form 1

```
class _import <class name>
class __import <class name>
```

Form 2

```
return_type _import <function name>
return_type __import <function name>
```

Form 3

```
data_type _import <data name>
data_type __import <data name>
```

Description

This keyword can be used as a class, function, or data modifier in 32-bit programs.

inline

Example Keywords

Syntax

```
inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the **inline** keyword to declare or define C++ inline functions.

Inline functions are best reserved for small, frequently used functions.

Example

```
inline char* cat_func(void) { return char*; }
```

int

See also Keywords

Syntax

```
[signed|unsigned] int <identifier> ;
```

Description

Use the **int** type specifier to define an integer data type.

Variables of type **int** can be signed (default) or unsigned.

long

See also Keywords

Syntax

```
long [int] <identifier> ;
[long] double <identifier> ;
```

Description

When used to modify an **int**, it doubles the number of bytes available to store the integer value.

When used to modify a **double**, it defines a floating-point data type with 80 bits of precision instead of 64.

The C++Builder IDE links the floating-point math package if you use floating-point values or operators anywhere in your program.

mutable

See also Example Keywords

Syntax

```
mutable <variable name>;
```

Description

Use the **mutable** specifier to make a variable modifiable even though it is in a **const**-qualified expression.

Using the mutable Keyword

Only class data members can be declared mutable. The **mutable** keyword cannot be used on **static** or **const** names. The purpose of **mutable** is to specify which data members can be modified by **const** member functions. Normally, a **const** member function cannot modify data members.

Example

```
#include <iostream.h>
class Alpha {
   mutable int count;
   mutable const int* iptr;
public:
   int func1(int i = 0) const { // Promises not to change const arguments.
      count = i++;  // But count can be changed.
      iptr = &i;
      cout << *iptr;
      return count;
      }
};

int main(void) {
   Alpha a;

   a.func1(0);
   return 0;
   }
```

operator

See also Example Keywords

Syntax

```
operator <operator symbol>( <parameters> )
{
   <statements>;
}
```

Description

Use the **operator** keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as *this is implicitly the first argument.

When you overload an operator as a friend, you can specify two arguments.

pascal, _pascal, __pascal

Keywords

Syntax

```
pascal <data-definition/function-definition> ;
_pascal <data-definition/function-definition> ;
__pascal <data-definition/function-definition> ;
```

Description

Use the **pascal**, **_pascal**, and **__pascal** keywords to declare a variable or a function using a Pascal-style naming convention (the name is in uppercase).

In addition, **pascal** declares Pascal-style parameter-passing conventions when applied to a function header (first parameter pushed first; the called function cleans up the stack).

In C++ programs, functions declared with the **pascal** modifer will still be mangled.

private

See also Keywords

Syntax

```
private: <declarations>
```

Description

A private member can be accessed only by member functions and friends of the class in which it is declared.

Class members are **private** by default.

You can override the default struct access with **private** or **protected** but you cannot override the default union access.

Friend declarations are not affected by these access specifiers.

protected

See also Keywords

Syntax

```
protected: <declarations>
```

Description

A protected member can be accessed by member functions and friends of the class in which it was declared, and by classes derived from the declared class.

You can override the default struct access with **private** or **protected** but you cannot override the default union access.

Friend declarations are not affected by these access specifiers.

public

See also Keywords

Syntax

```
public: <declarations>
```

Description

A public member can be accessed by any function.

Members of a struct or union are public by default.

You can override the default struct access with **private** or **protected** but you cannot override the default union access.

Friend declarations are not affected by these access specifiers.

__property

See also　　Keywords　　Example

Syntax

```
<property declaration> ::=
      __property <type> <id> [ <prop dim list> ] = "{" <prop attrib list>
  "}"

  <prop dim list> ::= "[" <type> [ <id> ] "]" [ <prop dim list> ]

  <prop attrib list> ::= <prop attrib> [ , <prop attrib list> ]

  <prop attrib> ::=     read = <data/function id>
  <prop attrib> ::=     write = <data/function id>
  <prop attrib> ::=     stored = <data/function id>
  <prop attrib> ::=     stored = <boolean constant>
  <prop attrib> ::=     default = <constant>
  <prop attrib> ::=     nodefault
  <prop attrib> ::=     index = <const int expression>
```

Description

Use **__property** to associate specialized read/write access with an identifier.

A property declaration in a class defines a named attribute for objects of the class and the actions associated with reading and modifying 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. A property can be any type except a file type.

Properties are a natural extension of data members in an object. Both can be used to express attributes of an object, but whereas data members are merely storage locations which can be examined and modified at will, properties provide greater control over *access* to attributes. Properties provide a mechanism for associating actions with the reading and writing of attributes, and they allow attributes to be *computed*.

For property arrays (when <*prop dim list*> is used), the index to the arrays can be of any type.

- Properties can only be declared in classes.
- The identifier in a read/write clause must be a data member or member function.

__published

See also Keywords

Syntax

```
__published: <declarations>
```

Description

Use the **__published** keyword to specify the properties that you want to be displayed in the Object Inspector. Only classes derived from *TObject* can have **__published** sections.

The visibility rules for published members are identical to those of public members. The only difference between published and public members is that Delphi-style run-time type information (RTTI) is generated for data members and properties that are declared in a **__published** section. RTTI enables an application to dynamically query the data members, member functions and properties of an otherwise unknown class type.

No constructors or destructors are allowed in a **__published** section. Properties, Pascal intrinsic or VCL derived data-members, member functions and closures are allowed.

register

Example Keywords

Syntax

```
register <data definition> ;
```

Description

Use the register storage class specifier to store the variable being declared in a CPU register (if possible), to optimize access and reduce code.

Items declared with the **register** keyword have a global lifetime.

Note: The C++Builder compiler can ignore requests for register allocation. Register allocation is based on the compiler’s analysis of how a variable is used.

Example

```
register int i;
```

return

Example Keywords

Syntax

```
return [ <expression> ] ;
```

Description

Use the **return** statement to exit from the current function back to the calling routine, optionally returning a value.

Example

```
double sqr(double x)
{
  return (x*x);
}
```

short

See also Example Keywords

Syntax

```
short int <variable> ;
```

Description

Use the short type modifier when you want a variable smaller than an int. This modifier can be applied to the base type int.

When the base type is omitted from a declaration, int is assumed.

Examples

```
short int i;
short     i;    /* same as "short int i;" */
```

signed

See also Example Keywords

Syntax

```
signed <type> <variable> ;
```

Description

Use the signed type modifier when the variable value can be either positive or negative. The signed modifier can be applied to base types **int**, **char**, **long** and **short**.

When the base type is omitted from a declaration, **int** is assumed.

Example

```
signed   int       i;    /* signed is default         */
signed             i;    /* same as "signed int i;"  */
unsigned long int  l;    /* int OK, not needed        */
signed   char      ch;   /* unsigned is default       */
```

The sizeof operator

See also Example Operators

The **sizeof** operator has two distinct uses:

sizeof *unary-expression*

sizeof (*type-name*)

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals `sizeof array/ sizeof array[0]` .

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

You cannot use **sizeof** with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is size_t.

You can use **sizeof** in preprocessor directives; this is specific to C++Builder.

In C++, `sizeof(classtype),` where *classtype* is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Example for sizeof operator

```
/*  USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>
struct st {
   char *name;
   int age;
   double height;
      };

struct st St_Array[]= {   /* AN ARRAY OF structs */
   { "Jr.",     4,   34.20 },   /* ST_Array[0] */
   { "Suzie",  23,   69.75 },   /* ST_Array[1] */
   };
int main() {
   long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };

   printf("\nNumber of elements in LD_Array = %d",
           sizeof(LD_Array) / sizeof(LD_Array[0]));

   /****  THE NUMBER OF ELEMENTS IN THE ST_Array. ****/
   printf("\nSt_Array has %d elements",
           sizeof(St_Array)/sizeof(St_Array[0]));

   /****  THE NUMBER OF BYTES IN EACH ST_Array ELEMENT.  ****/
   printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));

   /****  THE TOTAL NUMBER OF BYTES IN ST_Array.  ****/
      printf("\nSt_Array=%d", sizeof(St_Array));
   return 0;
   }
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 16
St_Array= 32
```

static

Example Keywords

Syntax

```
static <data definition> ;
static <function name> <function definition> ;
```

Description

Use the **static** storage class specifier with a local variable to preserve the last value between successive calls to that function. A **static** variable acts like a local variable but has the lifetime of an external variable.

In a class, data and member functions can be declared **static**. Only one copy of the **static** data exists for all objects of the class.

A **static** member function of a global class has external linkage. A member of a local class has no linkage. A **static** member function is associated only with the class in which it is declared. Therefore, such member functions cannot be **virtual**.

Static member functions can only call other **static** member functions and only have access to **static** data. Such member functions do not have a **this** pointer.

Examples

```
static int i;
static void printnewline(void) {}
```

_stdcall, __stdcall

Keywords

Syntax

```
__stdcall <function-name>
_stdcall <function-name>
```

Description

The **_stdcall** and **__stdcall** keywords force the compiler to generate function calls using the Standard calling convention. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments. Such functions comply with the standard WIN32 argument-passing convention.

Note: The **__stdcall** modifier is subject to name mangling.

struct

See also Example Keywords

Syntax

```
struct [<struct type name>] {
  [<type> <variable-name[, variable-name, ...]>] ;
    .
    .
    .
} [<structure variables>] ;
```

Description

Use a **struct** to group variables into a single record.

`<struct type name>`	An optional tag name that refers to the structure type.
`<structure variables>`	The data definitions, also optional.

Though both `<struct type name>` and `<structure variables>` are optional, one of the two must appear.

You define elements in the record by naming a `<type>`, followed by one or more `<variable-name>` (separated by commas).

Separate different variable types by a semicolon.

To access elements in a structure, use a record selector (.).

To declare additional variables of the same type, use the keyword **struct** followed by the `<struct type name>`, followed by the variable names.

Note: C++Builder allows the use of anonymous struct embedded within another structure.

Example

```
#include <string.h>
struct my_struct {
  char name[80], phone_number[80];
  int  age, height;
} my_friend;

void func() {
   strcpy(my_friend.name,"Mr. Wizard");   /* accessing an element */
}
struct my_struct my_friends[100];   /* declaring additional variables */
```

switch

See also Example Keywords

Syntax

```
switch ( <switch variable> ) {
  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;
}
```

Description

Use the switch statement to pass control to a case which matches the `<switch variable>`. At which point the statements following the matching case evaluate .

If no case satisfies the condition the default case evaluates.

To avoid evaluating any other cases and reliquish control from the switch, terminate each case with `break;`.

Example

```
/* Illustrates the use of keywords break, case, default, and switch. */
#include <conio.h>
#include <stdio.h>

int main(void) {
   int ch;

   printf("\tPRESS a, b, OR c. ANY OTHER CHOICE WILL "
          "TERMINATE THIS PROGRAM.");
   for ( /* FOREVER */; ((ch = getch()) != EOF); )
      switch (ch) {
         case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
            printf("\nOption a was selected.\n");
            break;
         case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
         case 'c' :
            printf("\nOption b or c was selected.\n");
            break;
         default :
            printf("\nNOT A VALID CHOICE!  Bye ...");
            return(-1);
         }
   return(0);
   }
```

this

See also Keywords

Syntax

```
class X {
  int a;
public:
  X (int b) {this -> a = b;}
```

Description

In nonstatic member functions, the keyword **this** is a pointer to the object for which the function is called. All calls to nonstatic member functions pass **this** as a hidden argument.

this is a local variable available in the body of any nonstatic member function. Use it implicitly within the function for member references. It does not need to be declared and it is rarely referred to explicitly in a function definition.

For example, in the call `x.func(y)` , where y is a member of X, the keyword **this** is set to &x and y is set to `this->y`, which is equivalent to x.y.

Static member functions do not have a **this** pointer because they are called with no particular object in mind. Thus, a static member function cannot access nonstatic members without explicitly specifying an object with . or ->.

throw

See also Keywords

Syntax

```
throw assignment-expression
```

Description

When an exception occurs, the throw expression initializes a temporary object of the type *T* (to match the type of argument *arg*) used in `throw(T arg)`. Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

_ _try

See also Keywords

Syntax

```
_ _try compound-statement handler-list
_ _try compound-statement termination-statement
```

Description

The **_ _try** keyword is supported only in C programs. Use try in C++ programs.

A block of code in which an exception can occur must be prefixed by the keyword **__try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the normal program flow is interrupted. The program begins a search for a handler that matches the exception. If the exception is generated in a C module, it is possible to handle the structured exception in either a C module or a C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

// try example

```
// In PROG.C
void func(void) {
   // generate an exception
   RaiseException( /* specify your arguments */ );
}

// In CALLER.CPP
// How to test for C++ or C-based exceptions.
#include <excpt.h>
#include <iostream.h>

int main(void) {
   try
   {             // test for C++ exceptions
      try
      {          // test for C-based structured exceptions
         func();
      }
      __except( /* filter-expression */ )
      {
      cout << "A structured exception was generated.";
      /* specify actions to take for this structured exception */
      return -1;
      }
      return 0;
   }
   catch ( ... )
   {
   // handler for any C++ exception
   cout << "A C++ exception was thrown.";
   return 1;
   }
}
```

try

See also Example Keywords

Syntax

```
try compound-statement handler-list
```

Description

The **try** keyword is supported only in C++ programs. Use __try in C programs.

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is tranferred to the handler

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

typedef

Example Keywords

Syntax

```
typedef <type definition> <identifier> ;
```

Description

Use the **typedef** keyword to assign the symbol name `<identifier>` to the data type definition `<type definition>`.

Examples

```
typedef unsigned char byte;
typedef char str40[41];
typedef struct {
     double re, im;
   } complex;
```

typename

Syntax 1

```
typename <identifier>
```

Syntax 2

```
template <  typename <identifier>  > class <identifier>
```

Description

Use the syntax 1 to reference a type that you have not yet defined. See example 1.

Use syntax 2 in place of the **class** keyword in a template declaration. See example 2.

typename Example 2

```
/* This example shows how the typename keyword can be used to replace the
  class keyword in a template declaration. */

#include <iostream.h>

template <typename T1, typename T2> T2 convert (T1 t1)
                             // use typename instead of class.
{ return (T2)t1; }

template <typename X, class Y> bool isequal (X x, Y y)
                             // mix typename and class.
{ if (x==y)return 1; return 0; }
```

typename Example 1

```
/* This example uses the typename keyword to declare variables as type T::A,
  which has not yet been defined. */

template <class T>
void f() {
  typedef typename T::A TA;   // declare TA as type T::A
  TA a5;                      // declare a5 as type TA
  typename T::A a6;           // declare a6 as type T::A
  TA * pta6;                  // declare pta6 as pointer to type TA
}
```

The typeid operator

See also Example Keywords

Syntax

```
typeid( expression )
typeid( type-name )
```

Description

You can use **typeid** to get run-time identification of types and expressions. A call to **typeid** returns a reference to an object of type **const** typeinfo. The returned object represents the type of the **typeid** operand.

If the **typeid** operand is a dereferenced pointer or a reference to a polymorphic type, **typeid** returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, **typeid** returns an object that represents the static type.

You can use the **typeid** operator with fundamental data types as well as user-defined types.

If the **typeid** operand is a dereferenced NULL pointer, the *Bad_typeid* exception is thrown.

// typeid example

```
// HOW TO USE operator typeid, Type_info::before(), AND Type_info::name()
#include <iostream.h>
#include <typeinfo.h>

class A { };
class B : A { };

void main() {
   char C;
   float X;

   // USE THE typeinfo::operator==()TO MAKE COMPARISON
   if (typeid( C ) == typeid( X ))
      cout << "C and X are the same type." << endl;
   else cout << "C and X are NOT the same type." << endl;

   // USE true AND false LITERALS TO MAKE COMPARISON
   cout << typeid(int).name();
   cout << " before " << typeid(double).name() << ": " <<
        (typeid(int).before(typeid(double)) ? true : false) << endl;
   cout << typeid(double).name();

   cout << " before " << typeid(int).name() << ": " <<
        (typeid(double).before(typeid(int)) ? true : false) << endl;

   cout << typeid(A).name();
   cout << " before " << typeid(B).name() << ": " <<
        (typeid(A).before(typeid(B)) ? true : false) << endl;
   }
```

Program Output

```
C and X are NOT the same type.
int before double: 0
double before int: 1
A before B: 1
```

union

See also   Example   Keywords

Syntax

```
union [<union type name>] {
  <type> <variable names> ;
  ...
} [<union variables>] ;
```

Description

Use unions to define variables that share storage space.

The compiler allocates enough storage in a_number to accommodate the largest element in the union.

Unlike a struct, the variables a_number.i and a_number.l occupy the same location in memory. Thus, writing into one overwrites the other.

Use the record selector (.) to access elements of a union .

Example

```
union int_or_long {
  int     i;
  long    l;
} a_number;
```

unsigned

See also Example Keywords

Syntax

```
unsigned <type> <variable> ;
```

Description

Use the **unsigned** type modifier when variable values will always be positive. The **unsigned** modifer can be applied to base types **int**, **char**, **long,** and **short**.

When the base type is omitted from a declaration, **int** is assumed.

Examples

```
unsigned int       i;
unsigned           i;    /* same as "unsigned int i;"    */
unsigned long int  l;    /* int OK, not needed           */
unsigned char      ch;   /* unsigned is default for char */
```

virtual

See also Keywords

Syntax

```
virtual class-name
virtual function-name
```

Description

Use the **virtual** keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as **virtual**, you can redefine it in any derived class, even if the number and type of arguments are the same.

The redefined function overrides the base class function.

void

Example Keywords

Syntax

```
void identifier
```

Description

void is a special type indicating the absence of any value. Use the **void** keyword as a function return type if the function does not return a value.

```
void hello(char *name)
{
  printf("Hello, %s.",name);
}
```

Use **void** as a function heading if the function does not take any parameters.

```
int init(void)
{
  return 1;
}
```

Void Pointers

Generic pointers can also be declared as **void**, meaning that they can point to any type.

void pointers cannot be dereferenced without explicit casting because the compiler cannot determine the size of the pointer object.

Example

```
int x;
float r;
void *p = &x;           /* p points to x */
int main (void)
{
  *(int *) p = 2;
  p = &r;               /* p points to r */
  *(float *)p = 1.1;
}
```

volatile

See also Keywords

Syntax

```
volatile <data definition> ;
```

Description

Use the **volatile** modifier to indicate that a variable can be changed by a background routine, an interrupt routine, or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment. It also prevents the compiler from making the variable a register variable

```
volatile int ticks;
void timer( ) {
   ticks++;
}
void wait (int interval) {
   ticks = 0;
   while (ticks < interval);  // Do nothing
}
```

The routines in this example (assuming *timer* has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop since the loop doesn't change the value of *ticks*.

Note: C++ extends **volatile** to include classes and member functions. If you've declared a **volatile** object, you can use only its **volatile** member functions.

while

See also Example Keywords

Syntax

```
while ( <condition> ) <statement>
```

Description

Use the **while** keyword to conditionally iterate a statement.

`<statement>` executes repeatedly until the value of `<condition>` is **false**. If no condition is specified, the **while** clause is equivalent to **while**(**true**).

The test takes place before `<statement>` executes. Thus, if `<condition>` evaluates to **false** on the first pass, the loop does not execute.

Example

```
while (*p == ' ') p++;
```

wchar_t (keyword)

Syntax

```
wchar_t <identifier>;
```

Description

In C++ programs, **wchar_t** is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. A **wchar_t** type is the same size, signedness, and alignment requirement as an **int** type.

C++ language support for the VCL

See also

Data types

Properties

Closures

OLE automation support

Open arrays

Exception handling support

Limitations

C++Builder supports the standard Borland C++ 5.01 language syntax for 32-bit applications. But C++Builder is heavily dependent on Delphi 2.01's Object Pascal based Visual Component Library (VCL) for building and running applications. Therefore, there are some necessary extensions and nuances in the C++ implementation which you should be aware of.

All references to VCL objects, and objects descended from VCL objects, are via pointers. VCL object references are normally made through the member access operator (**->**). For example, the following event handler in Delphi:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Listbox1.Items.Add(Edit1.Text);
  Edit1.Text := ‘’;
end;
```

In C++Builder this becomes:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Listbox1->Items->Add(Edit1->Text);
  Edit1->Text = “”;
}
```

The VCL is implemented using the **__fastcall** calling convention which passes function parameters and return values in registers. The IDE adds the **__fastcall** modifier to all functions and event handler code that it generates.

Support for Delphi data types

See also	Table of Delphi data types

C++Builder supports most Delphi intrinsic data types either by using a **typedef** to map the Delphi type to a standard C++ type, or by using a class to emulate the functionality of the type. The following table lists the Delphi data types and their C++Builder implementations.

Because most of the intrinsic Delphi types also exist in C++Builder, you have a choice of using Delphi-type declarations or C++Builder type declarations. For example, the line:

```
Single pi = 3.14159;
```

and the line:

```
float pi = 3.14159;
```

are both valid C++Builder syntax and equivalent.

Note: Remember, unlike Delphi, C++Builder is *case sensitive*. Therefore **Single** and **single** are not the same. Refer to the header file **include\vcl\sysdefs.h** for the correct spellings of Delphi types.

Table of Delphi data types

Delphi	Size/Values Implementation	C++ implementation	
ShortInt	8-bit integer	char	typedef
SmallInt	16-bit integer	short	typedef
LongInt	32-bit integer	long	typedef
Byte	8-bit unsigned integer	unsigned char	typedef
Word	16-bit unsigned integer	unsigned short	typedef
Integer	32-bit integer	int	typedef
Cardinal	32-bit unsigned integer	unsigned long	typedef
Boolean	true/false	bool	typedef
ByteBool	true/false or 8-bit unsigned integer	unsigned char	typedef
WordBool	true/false or 16-bit unsigned integer	unsigned short	typedef
LongBool	true/false or 32-bit unsigned integer	unsigned long	typedef
AnsiChar	8-bit unsigned character	unsigned char	typedef
WideChar	word-sized Unicode character	wchar_t	typedef
Char	8-bit unsigned character	char	typedef
AnsiString	Delphi AnsiString	AnsiString	class
String[n]	old style Delphi string, n = 1..255 bytes	SmallString<n>	template class
ShortString	old style Delphi string, 255 bytes	SmallString<255>	typedef
String	Delphi AnsiString	AnsiString	typedef
Single	32-bit floating point number	float	typedef
Double	64-bit floating point number	double	typedef
Extended	80-bit floating point number	long double	typedef
Currency	64-bit floating point number, 4 decimals	Currency	class
Real	32-bit floating point number	float	typedef
Comp	64-bit floating point number	double	typedef
Set	1..32 bytes	Set<type, minval, maxval>	template class
Pointer	32-bit generic pointer	void *	typedef
PChar	32-bit pointer to characters	unsigned char *	typedef
PAnsiChar	32-bit pointer to ANSI characters	unsigned char *	typedef
Variant	OLE variant value (16 bytes)	Variant	class

Special Delphi parameter types

See also

Some Delphi functions and procedures take parameter types that require special consideration. These parameter types include the following:

Open arrays	An array that does not explicitly state its element count.
Var parameters	A modifiable argument.
Untyped parameters	A parameter of unspecified type.

Open arrays

A Delphi open array uses the syntax:

```
array of <type>
```

For example:

```
Procedure MyFunc(x : array of integer);  // Call by value; size is not
  specified.
```

There is no intrinsic type in C++ that corresponds to the Pascal open array. However, its functionality can be obtained. Internally, when an **array of** *T* is passed as a parameter, it is broken down into a pointer to the array itself and a **long** that contains the highest element index in the array (or the number of elements of the array minus one). So in C++, the function prototype above becomes:

```
void MyFunc(int *x, long n);
```

where *x* is a pointer to the array of integers and *n* is passed the highest element index of the array.

For example, the VCL math function *Mean()* takes one parameter, an **array of Double**:

```
function Mean(const Data: array of Double): Extended;
```

Notice how the function is called in C++:

```
double d[4];
double *pd = &d;
d[0] =  3.14;
d[1] = 34.33;
d[2] =  9.34;
d[3] = 64.95;
// call by passing a pointer to the array and the highest element
long double x = Mean(pd, 3L);
```

var parameters

Functions that in Pascal take **var**, or modifiable parameters such as:

```
procedure myFunc(var x : int);
```

should be called using standard C++ “pass by reference” syntax:

```
void myFunc(int& x);
```

Untyped parameters

Pascal permits parameters to be passed to functions with no type defined. The receiving function must cast the parameter to a known type before using it. C++Builder takes untyped parameters as pointers-to-void (void *). The receiving function must cast the void pointer to a pointer of the desired type. For example:

```
int myfunc(void *MyName)
{
// Cast the pointer to the correct type; then dereference it.
return 1 + (int *)MyName;
}
```

Properties

See also Example

Use the **__property** keyword to associate specialized read/write access with an identifier.

Only classes can have properties. Properties look to a user like data members, but internally they can encapsulate member functions that read or write the value of the data member. A property definition in a class declares a named *attribute* for objects of the class, and actions associated with reading and writing the attribute. Examples of properties are the caption of a form, the size of a font or the name of a database table.

Properties let you create “side effects” for what appears to the user to be a simple data member. For example, in the C++Builder environment changing the *Caption* property of a form, which appears to be a data member of the form object, has the effect of immediately changing the title of the form window without the user explicitly calling a member function to do so.

A property declaration in a class defines a named attribute for objects of the class and the actions associated with reading and modifying the attribute. A property can be any type except a file type.

Properties are a natural extension of data members in an object. Both can be used to express attributes of an object, but whereas data members are merely storage locations which can be examined and modified at will, properties provide greater control over *access* to attributes. Properties provide a mechanism for associating actions with the reading and writing of attributes, and they allow attributes to be *computed*.

For VCL-based classes, properties also define the persistence of a class in which are stored and loaded when an object of the class is persistent.

For property arrays (when <*prop dim list*> is used), the index to the arrays can be of any type.

- Properties can only be declared in classes.
- The identifier in a read/write clause must be a data member or member function.

Property attributes

See also

Syntax

```
stored = <expression>
stored = <data member or member function>
default = <constant expression>
nodefault
index = <constant expression>
read = <data or member function>
write = <data or member function>
__dispid(constant int expression)
```

Description

stored = <*expression*> The value of the expression determines whether this property should be saved in the corresponding form file (.DFM) . If the *expression* is true, the property is saved.

default = <*constant expression*> This applies to ordinal type properties and hence the <*constant*> must be of an ordinal type. A property value is saved in a form file only if the current value is different from the default or there is no default value specified.

nodefault This is used by properties to override a previously declared *default*= value.

index = <*constant expression*> This is used when a single getting or setting routine is being used to support multiple properties. The index constant is a non-negative integer that is passed as an additional parameter to a read/write member function.

write = <*data or member function*> An access attribute of a property with an index attribute must list a member function name. In other words, the **read** and **write** attributes of a property with an index attribute are not allowed to list a data member.

When accessing a property with an index attribute, the integer value specified in the property definition is passed to the access function as an extra parameter. For that reason, an access function for a property with an index attribute must take an extra value parameter of type **int**.

The index is always passed as the last parameter. See the example for declaring indexed properties.

__dispid(*constant int expression)* A member function that has been declared in the **__automated** section of a class can include an optional **__dispid**(*constant int expression)* directive. The directive must be declared after the closing parenthesis of the parameter list.

Example Indexed properties

```
struct MyObject
{
   // This function is applied to two properties.
   // So, the index is used to distinguish Left and Right
   void Setter(double, int);
   __property double Left =
                   {write = Setter,
                    index = 0};
   __property double Right =
                    {write = Setter,
                     index = 1};
};
```

Property operators

See also

Arithmetic types

Properties of arithmetic types use compiler-defined operators. A property value can be changed by any of the usual assignment, increment, and decrement operators. If a getter function has been defined for a property, it is always used. For example, the following is acceptable.

```
property int i;
i++;
```

Incrementation is processed as follows.

```
i += 1;
```

Class types

For Delphi data types which are implemented as class types, C++Builder provides all the necessary operators.

Overloaded operators for properties of such class types require the creation of a temporary of that class. If you provide an overloaded operator, it will only be applied to the temporary class. You must still use the predefined assignment operator that is provided with the C++Builder implementation of that class type.

For example, if you overloaded the insertion operator for some property, you must use it as follows.

```
prop << 3;           // Updates a temporary, only.
prop = prop << 3;    // Updates the original.
```

Hoisted properties

See also Example

Syntax

```
__property <identifier> = { <attrib list> };
```

Description

A *hoisted property* is a property that was defined in a base class and has been modified in a derived class. Possible modifications can be changing the access rights of the properties or modifying selected attributes of the base class property.

Hoisted properties example

```
struct MyObject
{
   int Getter(void);
   bool IsStored(void);
__published:
   __property int myProperty = {
                     read = Getter,
                     stored = IsStored,
                     default = 42 };
};

struct MyDerivedObject : MyObject
{
__published:
   __property myProperty = { nodefault };
};
```

Property declarations

```
class myClass (
    int data1;
    int GetCounter();
    void SetCounter(int);
  public:
    myClass() {data1 = 0;};
    __property int counter = {read=GetCounter, write=SetCounter};
};
```

This class declares an integer property named *counter*. Whenever the value of *counter* is read, the *GetCounter* function is called. Therefore the following two statements are functionally equivalent:

```
int x = counter;
int x = GetCounter();
```

Likewise, when a value is assigned to *counter*, the function *SetCounter* will be called. Therefore the following two statements are functionally equivalent:

```
counter = 100;
SetCounter(100);
```

By using properties, the simple act of setting or getting a value can now have validation, error handling, or even modify values in the process of reading or writing. Setting values of properties can cause an action to take place beyond simply storing of a value.

Properties can also be tied to data members, not just functions. If, for example, the property in the above class had been declared as:

```
__property int counter = {read=data1, write=data1};
```

Then the statement:

```
int x = counter;
```

would assign the value of *data1* to *x*. This behavior could result in code such as:

```
counter = 10;     // assigns 10 to func1
data1 = 20;
int y = counter;  // y gets the value 20
```

Array properties

See also Example

Syntax

```
__property <type> <id> [<type>] = { <attrib list> };
```

Description

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. For example:

```
__property int widget[int index] = {read=GetWidget, write=SetWidget};
```

The format of an index parameter list is the same as a function's formal parameter list. Note that unlike array types, which can only specify integer dimensions, array properties allow indexes of any 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, just like an any array, can have more than one index. The corresponding parameters to the read and write functions must still have the same signatures as the indexes, and must appear in the same order as the indexes.

An access attribute of an array property must list a member function. In other words, the read and write attribute of an array property are not allowed to specify a data member.

The member functions listed in array property access attribute are governed by the following rules:

- The **read** attribute of an array property must list a function that takes the same number and types of parameters as in the property's index parameter list, and the function result type must be identical to the property type.
- The **write** attribute of an array property must list a function with the same number and types of parameters as are listed in the property's index parameter list, plus an additional parameter of the same type as the property type.

Array properties example

```
struct MyObject
{
int Getter(char *, int);
   void Setter(char *, int, int);
   __property int prop[char*][int] = { read = Getter, write = Setter);
};
int myFunc(MyObject * optr)
{
   return optr -> prop[“C++Builder”][1996];
}
```

Closures

See also Example

Use the **__closure** keyword to declare a pointer to a member function. Such pointers are referred to as *closures*.

A closure associates a pointer to a member function with a pointer to a class instance. The pointer to the class instance is used as the **this** pointer when calling the associated member function.

A closure declaration is the same as a function pointer declaration but with the addition of the **__closure** keyword before the identifier being defined. While a function pointer contains only a 4-byte code address, a closure contains both a code address (in the first 4 byte) and an object pointer (in the second 4 bytes) that serves as the **this** pointer when you call thru the closure.

Access specifiers

See also

In addition to the usual C++ access specifiers for classes, you can now define the area of your class which you want to be published in the Object Inspector or exposed to an OLE automation server. See the following topics.

Published properties

OLE automation support

Published properties

See also

Use the **__published** keyword to specify the properties that you want to be displayed in the Object Inspector when your class is a component that gets added to the component palette. Only classes derived from *TObject* can have **__published** sections.

The visibility rules for published members are identical to those of public members. The only difference between published and public members is that Delphi-style run-time type identification (RTTI) is generated for data members and properties that are declared in a **__published** section. RTTI enables an application to dynamically query the data members, member functions and properties of an otherwise unknown class type.

No constructors or destructors are allowed in a **__published** section. Properties, Pascal intrinsic or VCL derived data-members, member functions, and closures are allowed in a published section.

OLE automation support

See also Example

The base class for all objects in automation servers is TAutoObject.

The visibility rules for automated members are identical to those of public members. The only difference between automated and public components is that automation type information is generated for member functions and properties that are declared in an automated section. This automation information makes it possible to create OLE Automation servers.

For a member function, the types of all member function parameters and the function result (if any) must be automatable. Likewise, for a property, the property type and the types of any array property parameters must be automatable. Declaring member functions or properties that use non-automatable types in an **__automated** section results in an error.

The automatable types are:

Currency
double
int
float
short
String
TDateTime
Variant
unsigned short

- Classes should be derived from TAutoObject.
- Member function declarations must use the **__fastcall** calling convention.
- Member functions can be **virtual**.
- Member functions may add **__dispid**(*constant int expression*) after the closing parenthesis of the parameter list.
- Property declarations can only include access specifiers (**dispid**, **read** and **write**). No other specifiers (**index**, **stored**, **default**, **nodefault**) are allowed.
- Property access specifiers must list a member function identifier. Data member identifiers are not allowed.
- Property access member functions must use the **__fastcall** calling convention.
- Property overrides (property declarations that don’t include the property type) are not allowed.

Open arrays

See also

Temporary array arguments

Existing array arguments

Some VCL functions take open arrays as arguments. For such functions, you can pass the open array arguments with one of the following macros: EXISTINGARRAY, SLICE, or OPENARRAY.

A VCL function can require an open array argument in the following ways.

var	by reference, arguments can be changed
by value	the callee makes a copy of the arguments, the copy can be changed
	Very Important: If you write a C++ function that takes an open array by value as one of its arguments, the function must make a copy of the open array. The compiler does NOT do this automatically.
array of const	arguments cannot be changed. Always an array of type *TVarRec*.

The types of functions that take array arguments are of the following forms.

```
// Pascal function; call by value won’t modify its arguments
function sum(array of double): double;

// C++ equivalent of the function above; const argument won’t be modified
double sum(const double* nums, int sizeless1);

// This Pascal procedure can modify its argument
procedure zero(var array of double);
// C++ equivalent; the argument can be modified
void zero(double* nums, int sizeless1);
```

Temporary array arguments

See also Example

Temporary array arguments are required by Delphi functions and procedures that take an **array of T** but there is no argument that specifies the array element count. When calling such Delphi routines, you can construct open arrays as temporaries with the OPENARRAY macro. The macro is defined in INCLUDE\VCL\SYSDEFS.H header file. The declarative form is as follows.

```
// <T> is data type that the function is expecting.
OPENARRAY(<T>, (value1, value2, value3)) // Up to 19 values.
```

The OPENARRAY macro constructs a temporary of the specified type and initializes it with the values provided. You can specify as many as nineteen values. To specify more than nineteen values you must construct the array yourself.

When you use the OPENARRAY macro to construct temporary arrays, the macro guarantees proper deletion of the array.

Use the following guidelines when using temporary arrays as function arguments.

- Temporary arrays cannot be passed to a **var** open array.
- Temporary arrays can be passed by value to a function with an open array argument. The callee needs to make a copy of the temporary open array. The copy can be modified.

```
// The OPENARRAY macro constructs a temporary integer array.
OPENARRAY(int, (5, 7, 82, myIntVariable, 96))
```

- Temporary arrays can be used when a function argument is **array of const** type. The temporary array will be passed but it will not be modifed.

```
// You must always use TVarRec as the type.
OPENARRAY(TVarRec, (myString, fontName, 34, 22.0, myVariant, "abc", 'd'));
```

Existing array arguments

See also

When a VCL function requires an array, you can use an array of the matching type which you have defined. You can use the EXISTINGARRAY macro to pass such an array to a calling function. The EXISTINGARRAY macro provides the type and array size to the calling function.

Use the following guidelines when you want to pass an existing array as a function argument.

- You can pass an existing array to a function requiring a **var** open array.The existing array will be passed and it can be modified.

```
EXISTINGARRAY(myExistingArrayOfTheCorrectType)
```

- You can pass an existing array to a function that requires a value type open array as its argument. The callee will make a copy of the existing open array. The copy can be modified.

```
EXISTINGARRAY(myExistingArrayOfTheCorrectType)
```

- You can pass an existing array to a function that requires an **array of const** argument. Although technically this would work, it is unlikely that a preexisting *TVarRec* array would exist. You should not have *TVarRec* arrays in your application except when specifically required because they don't maintain proper reference counts (that is, pointers may point to nothing!)

```
// The example is not recommended usage.
EXISTINGARRAY(myExistingArrayOfTVarRecType)
```

Exception handling support for Delphi

See also

Operating system exceptions

Delphi exceptions

Portability considerations

The implementation of exception handling in C++Builder supports both, the Object Pascal mechanism as implemented in Delphi, and the C++ mechanism as implemented in Borland C++. When an exception is thrown from either mechanism, the usual, expected behaviour for that mechanism is invoked. In other words, the semantics for exception handling has not changed but it has been enlarged to allow both mechanisms to coexist.

There are some noteworthy differences between C++ and Delphi exception handling mechanisms.

Exceptions thrown from constructors:

- C++ destructors are called for members and base classes that are fully constructed.
- Object Pascal base class destructors are called even if the object or base class isn't fully constructed.

Catching exceptions:

- C++ exceptions can be caught by reference, pointer, or value. Exception derived from *TObject* can only be caught by reference or pointer. An attempt to catch *TObject* exceptions by value results in a compile-time error.
- Object Pascal exceptions are caught by reference.

Operating system exceptions

See also

C++Builder allows you to handle exceptions thrown by the operating system. Operating system exceptions include access violations, integer math errors, floating point math errors, stack overflow and control-c interrupts. These are handled in the C++ RTL and converted to Delphi-modeled exception class objects before being dispatched to your application. You can then write code in your C++ code that looks like this:

```
try
{
   char * p = 0;
   *p = 0;
}
// You should always catch by reference.
catch (const EAccessViolation &e)
{
   printf("You can't do that!\n");
}
```

The exception classes that the RTL uses here are necessarily non-ANSI because ANSI does not provide for this type of exception. The classes we use are precisely those that Delphi uses and are only available to C++Builder applications. They are derived from *TObject* and require the VCL underpinnings.

Here are some characteristics of the C++Builder exception-handling mechanism.

- The user is **not** responsible for freeing the exception object.
- Operating system exceptions can only be caught by pointer or reference. Catch by reference is the preferred approach.
- The user cannot rethrow an operating system exception once the catch frame has been exited and have it be caught by intervening Delphi catch frames.
- The user cannot rethrow an operating system exception once the catch frame has been exited and have it be caught by intervening operating system catch frames.

The last two points can be translated roughly as this: Once an operating system exception has been caught as a C++ exception, it cannot be rethrown as if it were an operating system exception or a Delphi exception if you are not in the catching stack frame.

Delphi exceptions

See also

C++Builder broadens the semantics for handling software exceptions thrown from Delphi or, equivalently, exceptions thrown from C++ where the exception class being thrown is derived from *TObject*. In such a case, there are a couple of rules that are derived from the fact that VCL-style classes can only be allocated on the heap.

- VCL-style exception classes may only be caught by pointer or reference (reference is preferred).
- VCL-style exception classes may only be thrown by pointer or reference.

Beyond this, there are no real limitations built in. C++Builder does not provide a mechanism for catching C++ exceptions in Delphi.

Portability considerations

See also

There are several RTLs being delivered with this product. Most of them pertain to C++Builder applications, but one of them (CW32MT.LIB) is the normal multithreaded RTL that does not make any references to VCL. This RTL is provided for support of legacy applications which may be part of a project but should not depend on VCL. This RTL does not have support for catching operating system exceptions because those exception objects are derived from *TObject* and would require that parts of VCL be linked into your application.

To get the full benefits of C++Builder, use the CP32MT.LIB library. This is the multithreaded runtime library that provides memory management and exception handling with the VCL.

Limitations

When passing open arrays to functions taking an “array of const" (that is, TVarRec*, int), you need to cast **float** and **double** values to **long double** (for example, `(long double)3.14159`).

The maximum number of open array elements that may be constructed as a temporary with the OPENARRAY macro is 19.

Examples

Example 1

Example 2

Examples

Example 1

Example 2

Example 3

TDateTime data type

See also Constructors Member functions Reference

Description

TDateTime is a C++ class that implements the Delphi *TDateTime* data type and the Delphi date-and-time runtime library routines that use the *TDateTime* data type.

The *TDateTime* class inherits a *Val* data member declared as a **double** that holds the date-time value. The integral part of a *TDateTime* value is the number of days that have passed since 12/30/1899. The fractional part of a *TDateTime* value is the time of day.

Following are some examples of *TDateTime* values and their corresponding dates and times:

0	12/30/1899 12:00 am
2.75	1/1/1900 6:00 pm
-1.25	12/29/1899 6:00 am
35065	1/1/1996 12:00 am

To find the fractional number of days between two dates, subtract the two values. To increment a date and time value by a certain fractional number of days, add the fractional number to the date and time value.

Note

Use only the operators within *TDateTime*. The compiler will ignore any operators you overload yourself.

TDateTime constructors

See also

Description

Use the constructors to create *TDateTime* objects from pointers to other *TDateTime* objects, **ints**, **doubles**, *AnsiStrings*, date arguments, or time arguments.

```
__fastcall TDateTime();
```

Constructs a *TDateTime* object with a date-time value of 0.

```
TDateTime(const TDateTimeBase& src);
TDateTime(const TDateTime& src);
```

Constructs a *TDateTime* object using the value of another *TDateTime* object. *TDateTimeBase* is the base class of *TDateTime*.

```
__fastcall TDateTime(const double src);
```

Constructs a *TDateTime* object from a **double** value.

```
__fastcall TDateTime(const int src);
```

Constructs a *TDateTime* object from an **int** value.

```
enum TDateTimeFlag {Date, Time, DateTime};
__fastcall TDateTime(const AnsiString& src, TDateTimeFlag flag = DateTime);
```

Constructs a *TDateTime* object from an *AnsiString*. The value of the *TDateTime* object is a date, time, or date-time, depending on the value of the *flag* argument.

```
__fastcall TDateTime(unsigned short year, unsigned short month,
    unsigned short day);
```

Constructs a *TDateTime* object from the date value specified with the *year*, *month*, and *day* arguments.

```
__fastcall TDateTime unsigned short hour, unsigned short min,
    unsigned short sec, unsigned short msec);
```

Constructs a *TDateTime* object from the time value specified with the hour, min, sec, and msec arguments

TDateTime data type reference

See also

Syntax

```
class __declspec(delphireturn) TDateTime : public TDateTimeBase
```

Constructors

```
__fastcall TDateTime()
__fastcall TDateTime(const TDateTimeBase& src)
__fastcall TDateTime(const TDateTime& src)
__fastcall TDateTime(const double src)
__fastcall TDateTime(const int src)
```

```
enum TDateTimeFlag {Date, Time, DateTime};
__fastcall TDateTime(const AnsiString& src, TDateTimeFlag flag = DateTime);
__fastcall TDateTime(unsigned short year, unsigned short month,
    unsigned short day);
__fastcall TDateTime(unsigned short hour, unsigned short min,
    unsigned short sec, unsigned short msec);
```

Operators

// Use only the operators within *TDateTime*.

// The compiler will ignore any operators you overload yourself.

```
TDateTime& __fastcall operator =(const TDateTimeBase& rhs)
TDateTime& __fastcall operator =(const TDateTime& rhs)
TDateTime& __fastcall operator =(const double rhs)
TDateTime& __fastcall operator =(const int rhs)
```

```
TDateTime& __fastcall operator +=(const TDateTimeBase& rhs)
TDateTime& __fastcall operator +=(const TDateTime& rhs)
TDateTime& __fastcall operator +=(const double rhs)
TDateTime& __fastcall operator +=(const int rhs)
```

```
TDateTime& __fastcall operator -=(const TDateTimeBase& rhs)
TDateTime& __fastcall operator -=(const TDateTime& rhs)
TDateTime& __fastcall operator -=(const double rhs)
TDateTime& __fastcall operator -=(const int rhs)
```

```
TDateTime& operator ++()
TDateTime operator ++(int)
TDateTime& operator --()
TDateTime operator --(int)
```

```
TDateTime __fastcall operator +(const TDateTimeBase& rhs) const
TDateTime __fastcall operator +(const TDateTime& rhs) const
TDateTime __fastcall operator +(const double rhs) const
TDateTime __fastcall operator +(const int rhs) const
```

```
TDateTime __fastcall operator -(const TDateTimeBase& rhs) const
TDateTime __fastcall operator -(const TDateTime& rhs) const
TDateTime __fastcall operator -(const double rhs) const
TDateTime __fastcall operator -(const int rhs) const
```

// comparisons

```
bool __fastcall operator ==(const TDateTime& rhs) const
bool __fastcall operator !=(const TDateTime& rhs) const
bool __fastcall operator >(const TDateTime& rhs) const
bool __fastcall operator <(const TDateTime& rhs) const
bool __fastcall operator >=(const TDateTime& rhs) const
bool __fastcall coperator <=(const TDateTime& rhs) const
```

```
__fastcall operator AncsiString() const;     //<Date||Time||
DateTime>String(smart)
__fastcall operator double() const
__fastcall operator int() const
```

Public member functions

```
static TDateTime __fastcall CurrentDate();
static TDateTime __fastcall CurrentTime();
static TDateTime __fastcall CurrentDateTime();
static TDateTime __fastcall FileDateToDateTime(int fileDate);
AnsiString __fastcall FormatString(const AnsiString& format);
AnsiString __fastcall DateString() const;
AnsiString __fastcall TimeString() const;
AnsiString __fastcall DateTimeString() const;
int __fastcall DayOfWeek() const;
int __fastcall FileDate() const;
```

TDateTime member functions

The following are the member functions of the *TDateTime* class.

CurrentDate()

CurrentTime()

CurrentDateTime()

DayOfWeek()

DateString()

DateTimeString()

DecodeDate()

DecodeTime()

FileDate()

FileDateToDateTime()

FormatString()

TimeString()

CurrentDate() member function

See also

Syntax

```
static TDateTime __fastcall CurrentDate();
```

Description

CurrentDate() returns the current date as a *TDateTime* value.

CurrentTime() member function

See also

Syntax

```
static TDateTime __fastcall CurrentTime();
```

Description

CurrentTime() returns the current time as a *TDateTime* value.

CurrentDateTime() member function

See also

Syntax

```
static TDateTime __fastcall CurrentDateTime();
```

Description

CurrentDateTime() returns the current date and time as a *TDateTime* value.

DayOfWeek() member function

See also

Syntax

```
int __fastcall DayOfWeek() const;
```

Description

DayOfWeek() returns the day of the week of the *TDateTime* value as an integer between 1 and 7. Sunday is the first day of the week and Saturday is the seventh.

DateString() member function

See also

Syntax

```
AnsiString __fastcall DateString() const;
```

Description

DateString() converts the date of the *TDateTime* value to a string. The conversion uses the format specified by the ShortDateFormat variable.

DateTimeString() member function

See also

Syntax

```
AnsiString __fastcall DateTimeString() const;
```

Description

The *DateTimeToStr()* converts the *TDateTime* value to a string. If the *TDateTime* value does not contain a date value, the date displays as 00/00/00. If the *TDateTime* value does not contain a time value, the time displays as 00:00:00 AM. You can change how the string is formatted by changing the value of one or more of the date and time formatting variables.

DecodeDate() member function

See also

Syntax

```
void __fastcall DecodeDate(unsigned short* year, unsigned short* month,
  unsigned short* day) const;
```

Description

DecodeDate() breaks apart the *TDateTime* value into year, month, and day values and stores these values in the *year*, *month*, and *day* parameters, respectively. Use *DecodeDate()* when you need to access the year, month, or day of a *TDateTime* value.

DecodeTime() member function

See also

Syntax

```
void __fastcall DecodeTime(unsigned short* hour, unsigned short* min,
  unsigned short* sec, unsigned short* msec) const;
```

Description

DecodeTime() breaks apart the *TDateTime* value into hour, minute, second, and millisecond values and stores these values in the *hour*, *min*, *sec*, and *msec* parameters, respectively. Use *DecodeDate()* when you need to access the hour, minute, second, or millisecond values of a *TDateTime* value.

FileDate() member function

See also

Syntax

```
int __fastcall FileDate() const;
```

Description

FileDate() converts the date-and-time value to a DOS date-and-time stamp.

FileDateToDateTime() member function

See also

Syntax

```
static TDateTime __fastcall FileDateToDateTime(int fileDate);
```

Description

FileDateToDateTime() converts a DOS file date-and-time value, specified with the *fileDate* argument, to a *TDateTime* value.

FormatString() member function

See also

Syntax

```
AnsiString __fastcall FormatString(const AnsiString& format);
```

Description

FormatString() returns the *TDateTime* value as a formatted string using the format specified by the *format* argument. The following format specifiers are supported:

Specifier	Displays
c	Displays the date using the format given by the ShortDateFormat variable, followed by the time using the format given by the LongTimeFormat variable. The time is not displayed if the fractional part of the *TDateTime* value is zero.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat) using the strings given by the ShortDayNames variable.
dddd	Displays the day as a full name (Sunday-Saturday) using the strings given by the LongDayNames variable.
ddddd	Displays the date using the format given by the ShortDateFormat variable.
dddddd	Displays the date using the format given by the LongDateFormat variable.
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec) using the strings given by the ShortMonthNames variable.
mmmm	Displays the month as a full name (January-December) using the strings given by the LongMonthNames variable.
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
t	Displays the time using the format given by the ShortTimeFormat variable.
tt	Displays the time using the format given by the LongTimeFormatvariable.
am/pm	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
a/p	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower,

	upper, or mixed case, and the result is displayed accordingly.
ampm	Uses the 12-hour clock for the preceding h or hh specifier, and displays the contents of the TimeAMString variable for any hour before noon, and the contents of the TimePMString variable for any hour after noon.
/	Displays the date separator character given by the DateSeparator variable.
:	Displays the time separator character given by the TimeSeparatorvariable.
'xx'/"xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

Format specifiers may be written in upper case as well as in lower case letters. Both produce the same result.

If the string given by the format parameter is empty, the date and time value is formatted as if a 'c' format specifier had been given.

TimeString() member function

See also

Syntax

```
AnsiString __fastcall TimeString() const;
```

Description

TimeToStr() converts the *TDateTime* value to a string. The conversion uses the format specified by the LongTimeFormat variable. You can change the format of how the string is displayed by changing the value of one or more of the date-and time-formatting variables.

TDateTime formatting variables

See also

Syntax

```
extern char DateSeparator;
extern System::AnsiString ShortDateFormat;
extern System::AnsiString LongDateFormat;
extern char TimeSeparator;
extern System::AnsiString TimeAMString;
extern System::AnsiString TimePMString;
extern System::AnsiString ShortTimeFormat;
extern System::AnsiString LongTimeFormat;
extern System::AnsiString ShortMonthNames[12];
extern System::AnsiString LongMonthNames[12];
extern System::AnsiString ShortDayNames[7];
extern System::AnsiString LongDayNames[7];
```

Description

SYSUTILS.HPP includes a number of variables that are used by the *TDateTime* class. You can assign new values to these variables to change the formats of date and time strings. The initial values of these variables are fetched from the system registry using *GetLocaleInfo()* in the Win32 API. C++Builder applications automatically update these formatting variables in response to *WM_WININICHANGE* messages.

Application->UpdateFormatSettings() either allows or disallow changes in system settings. The default is **true**. Set this property to **false** to prevent the system settings from changing.

The description of each variable specifies the *LOCALE_XXXX* constant used to fetch the initial value using the *GetLocaleInfo()* Win32 API.

Variable	Defines
DateSeparator	*DateSeparator* is the character used to separate the year, month, and day parts of a date value. The initial value is fetched from *LOCATE_SDATE*.
ShortDateFormat	*ShortDateFormat* is the format string used to convert a date value to a short string suitable for editing. For a complete description of date and time format strings, refer to the documentation for the *FormatString()* function. The short date format should only use the date separator character and the m, mm, d, dd, yy, and yyyy format specifiers. The initial value is fetched from *LOCALE_SSHORTDATE*.
LongDateFormat	*LongDateFormat* is the format string used to convert a date value to a long string suitable for display but not for editing. For a complete description of date and time format strings, refer to the documentation for the *FormatString()* function. The initial value is fetched from *LOCALE_SLONGDATE*.
TimeSeparator	TimeSeparator is the character used to separate the hour, minute, and second parts of a time value. The initial value is fetched from *LOCALE_STIME*.
TimeAMString	*TimeAMString* is the suffix string used for time values between 00:00 and 11:59 in 12-hour clock format. The initial value is fetched from *LOCALE_S1159*.
TimePMString	*TimePMString* is the suffix string used for time values between 12:00 and 23:59 in 12-hour clock format. The initial value is fetched from *LOCALE_S2359*.
ShortTimeFormat	*ShortTimeFormat* is the format string used to convert a time value to a short string with only hours and minutes. The default value is computed from *LOCALE_ITIME* and *LOCALE_ITLZERO*.
LongTimeFormat	*LongTimeFormat* is the format string used to convert a time value to a long string with hours, minutes, and seconds. The default value is computed from

	LOCALE_ITIME and *LOCALE_ITLZERO*.
ShortMonthNames	*ShortMonthNames* is the array of strings containing short month names. The mmm format specifier in a format string passed to *FormatString()* causes a short month name to be substituted. The default values are fetched from the *LOCALE_SABBREVMONTHNAME* system locale entries.
LongMonthNames	*LongMonthNames* is the array of strings containing long month names. The mmmm format specifier in a format string passed to *FormatString()* causes a long month name to be substituted. The default values are fetched from the *LOCALE_SMONTHNAME* system locale entries.
ShortDayNames	*ShortDayNames* is the array of strings containing short day names. The ddd format specifier in a format string passed to *FormatString()* causes a short day name to be substituted. The default values are fetched from the *LOCALE_SABBREVDAYNAME* system locale entries.
LongDayNames	*LongDayNames* is the array of strings containing long day names. The dddd format specifier in a format string passed to *FormatString()* causes a long day name to be substituted. The default values are fetched from the *LOCALE_SDAYNAME* system locale entries.

Currency data type

See also Constructors Reference

Description

Currency is a C++ class that implements the Delphi *Currency* data type.

Currency inherits a *Val* data member declared as **int64** that holds the currency value. The range of possible currency values is -922337203685477.5808 to 922337203685477.5807. Use *Currency* hold monetary values.

Note

Use only the operators within *Currency*. The compiler will ignore any operators you overload yourself.

Currency data type reference

See also

Syntax

```
class __declspec(delphireturn) Currency : public CurrencyBase
```

Constructors

```
__fastcall Currency()
__fastcall Currency(double val)
__fastcall Currency(int val)
__fastcall Currency(const CurrencyBase& src)
__fastcall Currency(const Currency& src)
__fastcall Currency(const AnsiString& src);
```

Operators

// Use only the operators within *Currency*.

// The compiler will ignore any operators you overload yourself.

```
friend Currency __fastcall operator +(int lhs, const Currency& rhs);
friend Currency __fastcall operator -(int lhs, const Currency& rhs);
friend Currency __fastcall operator *(int lhs, const Currency& rhs);
friend Currency __fastcall operator /(int lhs, const Currency& rhs);
friend Currency __fastcall operator +(double lhs, const Currency& rhs);
friend Currency __fastcall operator -(double lhs, const Currency& rhs);
friend Currency __fastcall operator *(double lhs, const Currency& rhs);
friend Currency __fastcall operator /(double lhs, const Currency& rhs);

Currency& __fastcall operator =(double rhs)
Currency& __fastcall operator =(int rhs)
Currency& __fastcall operator =(const CurrencyBase& rhs)
Currency& __fastcall operator =(const Currency& rhs)

Currency& __fastcall operator +=(const Currency& rhs)
Currency& __fastcall operator -=(const Currency& rhs)
Currency& __fastcall operator *=(const Currency& rhs)
Currency& __fastcall operator /=(const Currency& rhs)
Currency& __fastcall operator %=(int rhs)

Currency& operator ++()
Currency operator ++(int)
Currency& operator --()
Currency operator --(int)

Currency __fastcall operator +(const Currency& rhs) const
Currency __fastcall operator -(const Currency& rhs) const
Currency __fastcall operator *(const Currency& rhs) const
Currency __fastcall operator /(const Currency& rhs) const

Currency __fastcall operator +(int rhs) const
Currency __fastcall operator -(int rhs) const
Currency __fastcall operator *(int rhs) const
Currency __fastcall operator /(int rhs) const
Currency __fastcall operator %(int rhs) const

Currency __fastcall operator +(double rhs) const
Currency __fastcall operator -(double rhs) const
Currency __fastcall operator *(double rhs) const
Currency __fastcall operator /(double rhs) const
Currency __fastcall operator -() const

Currency __fastcall operator !() const
```

// comparisons (Currency rhs)

```
bool __fastcall operator ==(const Currency& rhs) const
bool __fastcall operator !=(const Currency& rhs) const
bool __fastcall operator >(const Currency& rhs) const
bool __fastcall operator <(const Currency& rhs) const
bool __fastcall operator >=(const Currency& rhs) const
bool __fastcall operator <=(const Currency& rhs) const
```

// comparisons (int rhs)

```
bool __fastcall operator ==(int rhs) const
bool __fastcall operator !=(int rhs) const
bool __fastcall operator >(int rhs) const
bool __fastcall operator <(int rhs) const
bool __fastcall operator >=(int rhs) const
bool __fastcall operator <=(int rhs) const
```

// comparisons (double rhs)

```
bool __fastcall operator ==(double rhs) const
bool __fastcall operator !=(double rhs) const
bool __fastcall operator >(double rhs) const
bool __fastcall operator <(double rhs) const
bool __fastcall operator >=(double rhs) const
bool __fastcall operator <=(double rhs) const

__fastcall operator double() const
__fastcall operator int() const
__fastcall operator AnsiString() const;
```

```
inline Currency __fastcall operator +(int lhs, const Currency& rhs)
inline Currency __fastcall operator -(int lhs, const Currency& rhs)
inline Currency __fastcall operator *(int lhs, const Currency& rhs)
inline Currency __fastcall operator /(int lhs, const Currency& rhs)
inline Currency __fastcall operator +(double lhs, const Currency& rhs)
inline Currency __fastcall operator -(double lhs, const Currency& rhs)
inline Currency __fastcall operator *(double lhs, const Currency& rhs)
inline Currency __fastcall operator /(double lhs, const Currency& rhs)

inline ostream& operator <<(ostream& os, const Currency& arg)
inline istream& operator >>(istream& is, Currency& arg)
```

Currency constructors

See also

Description

Use the *Currency* constructors to create *Currency* objects from pointers to other *Currency* objects, **ints**, **doubles**, or *AnsiString* values.

```
__fastcall Currency();
```

Constructs a *Currency* object with a currency value of 0.

```
__fastcall Currency(double val);
```

Constructs a *Currency* object from a **double** value.

```
__fastcall Currency(int val);
```

Constructs a *Currency* object from an **int** value.

```
__fastcall Currency(const CurrencyBase& src);
__fastcall Currency(const Currency& src);
```

Constructs a *Currency* object using the value of another *Currency* object. *CurrencyBase* is the base class of *Currency*; it contains only the *Val* data member, which holds the currency value.

```
__fastcall Currency(const AnsiString& src);
```

Constructs a *Currency* object from an *AnsiString*.

Currency formatting variables

See also

Syntax

```
extern System::AnsiString CurrencyString;
extern unsigned char CurrencyFormat;
extern unsigned char NegCurrFormat;
extern char ThousandSeparator;
extern char DecimalSeparator;
extern unsigned char CurrencyDecimals;
```

Description

SYSUTILS.HPP includes a number of variables that are used by the *Currency* class. You can assign new values to these variables to change the formatting of *Currency* values. The initial values of these variables are fetched from the system registry using *GetLocaleInfo()* in the Win32 API. C++Builder applications automatically update these formatting variables in response to *WM_WININICHANGE* messages.

Application->UpdateFormatSettings() either allows or disallow changes in system settings. The default is **true**. Set this property to **false** to prevent the system settings from changing.

The description of each variable specifies the *LOCALE_XXXX* constant used to fetch the initial value using the *GetLocaleInfo()* Win32 API.

Variable	Defines
CurrencyString	*CurrencyString* defines the currency symbol (or characters) used in floating-point to decimal conversions. The initial value is fetched from *LOCALE_SCURRENCY*.
CurrencyFormat	*CurrencyFormat* defines the currency symbol placement and separation used in floating-point to decimal conversions. Possible values are: 0 = '$1' 1 = '1$' 2 = '$ 1' 3 = '1 $' The initial value is fetched from *LOCALE_ICURRENCY*.
NegCurrFormat	*NegCurrFormat* defines the currency format used in floating-point to decimal conversions of negative numbers. Possible values are: 0 = ($1) 8 = -1 $ 1 = -$1 9 = -$ 1 2 = $-1 10 = 1 $- 3 = $1- 11 = $ 1- 4 = (1$) 12 = $ -1 5 = -1$ 13 = 1- $ 6 = 1-$ 14 = ($ 1) 7 = 1$- 15 = (1 $) The initial value is fetched from *LOCALE_INEGCURR*.
ThousandSeparator	*ThousandSeparator* is the character used to separate thousands in numbers with more than three digits to the left of the decimal separator. The initial value is fetched from *LOCALE_STHOUSAND*.
DecimalSeparator	*DecimalSeparator* is the character used to separate the integer part from the

	fractional part of a number. The initial value is fetched from *LOCALE_SDECIMAL*.
CurrencyDecimals	*CurrencyDecimals* is the number of digits to the right of the decimal point in a currency amount. The initial value is fetched from *LOCALE_ICURRDIGITS*.

Extended Delphi data types

See also

Some complex data types used in Object Pascal and Delphi cannot be simply **typedef**’ed in C++Builder. Support for such data types is provided in C++Builder by implementing them as C++ classes. The extended Delphi data types are defined in INCLUDE\VCL\SYSDEFS.H header file. Each class provides all the necessary constructors, member functions, and operators.

Use the C++Builder implementation of these data types when you use the VCL or write your own components.

The extended Delphi data types that are defined as C++ classes include the following.

Set

AnsiString

Variant

TDateTime

Currency

Set data type

See also Example Reference

Use the *Set* type as defined in INCLUDE\VCL\SYSDEFS.H to define types used as VCL function parameters or VCL function return types.

The `Set<type, minval, maxval>` template class is a C++ class that implements the Delphi intrinsic type **set**. You must specify three template parameters:

- *type*: the type of the set elements (usually **int**, **char**, or an **enum** type)
- *minval*: the minimum value the set can hold (this value cannot be less then 0)
- *maxval*: the maximum value the set can hold (this value cannot be greater than 255)

Each instantiation of a *Set* creates an object based on all three parameters. Therefore, the following are two distinct types:

```
Set <char, 'A', 'C'> s1;
Set <char, 'X', 'Z'> s2;
if (s1 == s2)   // ERROR; illegal struct.
```

To create multiple instances of a *Set* type, use a **typedef** expression.

```
typedef Set <char, 'A','Z'> UPPERCASESet;
```

The declaration of a *Set* variable does *not* initialize the variable. You can declare the set types and initialize by using this syntax.

```
UPPERCASESet s1;
s1 << 'A' << 'B' << 'C';   // Initialize

UPPERCASESet s2;
s2 << 'X' << 'Y' << 'Z';   // Initialize
```

Set

See also

Syntax

```
template<class T, unsigned char minEl, unsigned char maxEl>
class __declspec(delphireturn) Set;
```

Public constructor

```
__fastcall Set();
__fastcall Set(const Set& src);
```

Public member functions

```
bool __fastcall Contains(const T el) const;
Set& __fastcall Clear();
```

Public operators

```
Set& __fastcall operator =(const Set& rhs);
Set& __fastcall operator +=(const Set& rhs); //Union
Set& __fastcall operator -=(const Set& rhs); //Difference
Set& __fastcall operator *=(const Set& rhs); //Intersection
Set __fastcall operator +(const Set& rhs) const; //Union
Set __fastcall operator -(const Set& rhs) const; //Difference
Set __fastcall operator *(const Set& rhs) const; //Intersection
Set& __fastcall operator <<(const T el); //Add element
Set& __fastcall operator >>(const T el); //Remove element
bool __fastcall operator ==(const Set& rhs) const;
bool __fastcall operator !=(const Set& rhs) const ;
```

Example

```
#include <sysdefs.h>
#include <iostream.h>

typedef Set<char,'U','Z'> UpperSet;
typedef Set<char,'a','z'> LowerSet;
typedef Set<char,'a','j'> HalfLowerSet;


void set_example()
{
LowerSet ae, ae2, ac, de;
UpperSet AE, AE2, AC, DE;
HalfLowerSet aj;

// Both of these are false. Sets are empty until members are added.
cout <<"Set ae " << (ae.Contains('a')?"does":"does not") << " contain 'a' "
  << endl;
cout <<"Set AE " << (AE.Contains('C')?"does":"does not") << " contain 'C' "
  << endl;

ae  << 'a' << 'b' << 'c' << 'd' << 'e';
ae2 << 'a' << 'b' << 'c' << 'd' << 'e';
ac  << 'a' << 'b' << 'c';
de  << 'd' << 'e';
aj  << 'd' << 'e';
DE  << 'D' << 'E';

cout <<"Now, set ae " << (ae.Contains('a')?"does":"does not") << " contain
  'a' " << endl;

// Compiler-time error! Although sets 'de' and 'aj' contain the same
  members,
// they are different types of sets and cannot be compared.
//cout << (de==aj?" == ":" != ") << endl;

// Sets support operations * (intersection),+ (union) and - (difference).
cout << "Set ae " << (ae==(ac+de)?" == ":" != ") << "set ac + de." << endl;
cout << "Set de " << (de==(ae-ac)?" == ":" != ") << "set ae - ac." << endl;

// Clear member function
cout << "Set ae2: " << ae2 << endl;
ae2.Clear();
cout << "Set ae2: " << ae2 << " after ae2.Clear()" << endl;

// Sets also support operations *= (intersection),+= (union) and -=
  (difference).
ae2+=ac;
cout << "Union ae2+=ac: Set ae2 " << (ae2==ac?" == ":" != ") << "set ac." <<
  endl;
ae2+=de;
cout << "Union ae2+=de: Set ae2 " << (ae2==ae?" == ":" != ") << "set ae." <<
  endl;
ae2-=ac;
cout << "Difference ae2-=ac: Set ae2 " << (ae2==de?" == ":" != ") << "set
  de." << endl;
}

int main()
{
   set_example();

return 0;
}
```

AnsiString data type

See also Example Reference

C++Builder implements the *AnsiString* type as a class. *AnsiString* is designed to function like the Delphi long string type. Accordingly, *AnsiString* provides the following string handling characteristics which are required when you call VCL-type functions that use any of the Delphi long string types.

- reference count
- string length
- data
- null string terminator

If you don’t provide an initial value, *AnsiString* variables are zero-initialized upon instantiation.

Example

```
/* Compile with  bcc32 famille.cpp vcl.lib ole2w32.lib */
#include <vcl/dstring.h>
#include <stdio.h>

class Famille
{
private:
   AnsiString FNames[10];
   AnsiString GetName(int Index);
   void SetName(int, AnsiString);
public:
    __property AnsiString Names[int Index] = {read=GetName, write=SetName};
   Famille(){}
   ~Famille(){}
};

AnsiString Famille::GetName(int i)
{
   return FNames[i];
}

void Famille::SetName(int i,const AnsiString s)
{
   FNames[i]=s;
}

int main()
{
   Famille C;
   C.Names[0]="Steve";   //calls Famille::SetName()
   C.Names[1]="Amy";
   C.Names[2]="Sarah";
   C.Names[3]="Andrew";
   for (int i = 0; i <= 3; i++)
   {
      //calls Famille::GetName()
      puts(C.Names[i].c_str());
   }
}
```

AnsiString

Syntax
```
class __declspec(delphireturn) AnsiString
```

Public constructors
```
__fastcall AnsiString();
```
Creates an empty string.

```
__fastcall AnsiString(const char* src);
__fastcall AnsiString(const AnsiString& src);
__fastcall AnsiString(const char* src, unsigned char len);
__fastcall AnsiString(const wchar_t* src);
__fastcall AnsiString(int src);
__fastcall AnsiString(double src);
```

Public destructor
```
__fastcall ~AnsiString();
```

Public data member
```
enum TStringFloatFormat {sffGeneral, sffExponent, sffFixed, sffNumber,
  sffCurrency };
```
The TStringFloatFormat enum is used by FloatToStrF

Operators
```
friend AnsiString __fastcall operator +(const char*, const AnsiString& rhs);
static AnsiString __fastcall StringOfChar(char ch, int count);
static AnsiString __fastcall LoadStr(int ident);
static AnsiString __fastcall FmtLoadStr(int ident, const TVarRec *args, int
  size);
static AnsiString __fastcall Format(const AnsiString& format,const TVarRec
  *args, int size);
static AnsiString __fastcall FormatFloat(const AnsiString& format,const long
  double& value);
static AnsiString __fastcall FloatToStrF(long double value,
  TStringFloatFormat format, int precision, int digits);
static AnsiString __fastcall IntToHex(int value, int digits);
static AnsiString __fastcall CurrToStr(Currency value);
static AnsiString __fastcall CurrToStrF(Currency value, TStringFloatFormat
  format, int digits);

    // Assignments
    AnsiString& __fastcall operator =(const AnsiString& rhs);
    AnsiString& __fastcall operator +=(const AnsiString& rhs);

    //Comparisons
    bool __fastcall operator ==(const AnsiString& rhs) const;
    bool __fastcall operator !=(const AnsiString& rhs) const;
    bool __fastcall operator <(const AnsiString& rhs) const;
    bool __fastcall operator >(const AnsiString& rhs) const;
    bool __fastcall operator <=(const AnsiString& rhs) const;
    bool __fastcall operator >=(const AnsiString& rhs) const;
    int __fastcall AnsiCompare(const AnsiString& rhs) const;
    int __fastcall AnsiCompareIC(const AnsiString& rhs) const; //ignorecase

    // Index
    char& __fastcall operator [](const int idx) {return Data[idx];}

    // Concatenation
    AnsiString __fastcall operator +(const AnsiString& rhs) const;

    // C string operator
    char* __fastcall c_str() const
      {return (Data)? Data: "";}

int __fastcall Length() const;
bool __fastcall IsEmpty() const;

    // make string unique (refcnt == 1)
    void __fastcall Unique();

    void __fastcall Insert(const AnsiString& str, int index);
    void __fastcall Delete(int index, int count);
    void __fastcall SetLength(int newLength);

    int __fastcall Pos(const AnsiString& subStr) const;
    AnsiString __fastcall LowerCase() const;
    AnsiString __fastcall UpperCase() const;
    AnsiString __fastcall Trim() const;
    AnsiString __fastcall TrimLeft() const;
    AnsiString __fastcall TrimRight() const;
    AnsiString __fastcall SubString(int index, int count) const;

    int __fastcall ToInt() const;
    int __fastcall ToIntDef(int defaultValue) const;
    double __fastcall ToDouble() const;

    //Convert to Unicode
    int __fastcall WideCharBufSize() const;
    wchar_t* __fastcall WideChar(wchar_t* dest, int destSize) const;

#if defined(MBCS)
    // mbcs support
    enum TStringMbcsByteType { mbSingleByte, mbLeadByte, mbTrailByte };

    TStringMbcsByteType __fastcall ByteType(int index) const;
    bool __fastcall IsLeadByte(int index) const;
    bool __fastcall IsTrailByte(int index) const;
    bool __fastcall IsDelimiter(const AnsiString& delimiters, int index)
  const;
    bool __fastcall IsPathDelimiter(int index) const;
    int __fastcall LastDelimiter(const AnsiString& delimiters) const;
    int __fastcall AnsiPos(const AnsiString& subStr) const;
    char* __fastcall AnsiLastChar() const;
#endif
```

Variant data type

See also Reference

The *Variant* class is a C++Builder implementation of the Delphi intrinsic type **Variant**.

In C++Builder, the syntax for using variants is different from the Delphi usage. For example, if you have the following Delphi code:

```
V: Variant;
V := VarArrayCreate([0,HighVal,0,HighVal],varInteger);
```

In C++Builder you can use *Variant* and *OPENARRAY* like this.

```
Variant V(OPENARRAY(int,(0,HighVal,0,HighVal)),varInteger);
```

Variant

See also

Syntax

```
class __declspec(delphireturn) Variant: public TVarData
```

Public constructors

```
__fastcall Variant();
__fastcall Variant(const Variant& src);

//By value constructors
__fastcall Variant(const short src);
__fastcall Variant(const int src);
__fastcall Variant(const float src);
__fastcall Variant(const double src);
__fastcall Variant(const Currency src);
__fastcall Variant(const TDateTime src);
__fastcall Variant(const bool src);
__fastcall Variant(const WordBool src);
__fastcall Variant(const Byte src);
__fastcall Variant(const AnsiString& src);
__fastcall Variant(const char* src); // treat as asciiz pointer
__fastcall Variant(wchar_t* const src);
__fastcall Variant(Ole2::IDispatch* const src);
__fastcall Variant(Ole2::IUnknown* const src);

//By ref constructors
__fastcall Variant(short* src);
__fastcall Variant(int* src);
__fastcall Variant(float* src);
__fastcall Variant(double* src);
__fastcall Variant(Currency* src);
__fastcall Variant(TDateTime* src);
__fastcall Variant(WordBool* src);
__fastcall Variant(Byte* src);
__fastcall Variant(wchar_t** src);

// constructor for array of variants of type varType
__fastcall Variant(const int* bounds, const int boundsSize, Word varType);

// constructor for one dimensional array of type Variant
__fastcall Variant(const Variant* values, const int valuesSize);
```

Operators

```
~ Variant &
-    (int negation)
+ Variant &
- Variant &
* Variant &
/ Variant &
% Variant &
<< Variant &
>> Varian-++++++t &
| Variant &
& Variant &
^ Variant &
```

```
= Variant &
+= Variant &
-= Variant &
*= Variant &
/= Variant &
%= Variant &
<<= Variant &
>>= Varinat &
|= Variant &
&= Variant &
^ Variant &
```

Public member functions

```
// HRESULT methods
void __fastcall SetError(const Integer err);
Integer __fastcall GetError() const;

void __fastcall Clear();
Variant& __fastcall ChangeType(int VarType);
Variant __fastcall AsType(int VarType) const;

int __fastcall Type() const;
bool __fastcall IsNull() const;
bool __fastcall IsEmpty() const;

// variant array stuff
bool    __fastcall IsArray() const;
Variant __fastcall GetElement(const int i1) const;
Variant __fastcall GetElement(const int i1, const int i2) const;
Variant __fastcall GetElement(const int i1, const int i2, const int i3)
  const;
Variant __fastcall GetElement(const int i1, const int i2, const int i3,
  const int i4) const;
Variant __fastcall GetElement(const int i1, const int i2, const int i3,
  const int i4, const int i5) const;
void __fastcall PutElement(const Variant& data, const int i1);
void __fastcall PutElement(const Variant& data, const int i1, const int i2);
void __fastcall PutElement(const Variant& data, const int i1, const int i2,
  const int i3);
void __fastcall PutElement(const Variant& data, const int i1, const int i2,
  const int i3, const int i4);
void __fastcall PutElement(const Variant& data, const int i1, const int i2,
  const int i3, const int i4, const int i5);
int __fastcall ArrayDimCount() const;
int __fastcall ArrayLowBound(const int dim = 1) const;
int __fastcall ArrayHighBound(const int dim = 1) const;
void __fastcall ArrayRedim(int highBound);
Pointer __fastcall ArrayLock();
void    __fastcall ArrayUnlock();

// Automation Goodies
Variant __fastcall Exec(AutoCmd& cmd, Integer lcid = LOCALE_SYSTEM_DEFAULT);

// Alternate Syntax for Automation
// (Doesn't support Named Parameters for now)
void OleProcedure(const String& name, Variant& v0 = Variant(),
  Variant& v1 = Variant(),Variant& v2 = Variant(),Variant& v3 = Variant(),
  Variant& v4 = Variant(),Variant& v5 = Variant(),Variant& v6 = Variant(),
  Variant& v7 = Variant(),Variant& v8 = Variant(),Variant& v9 = Variant());
Variant OleFunction(const String& name, Variant& v0 = Variant(),
   Variant& v1 = Variant(),Variant& v2 = Variant(),Variant& v3 = Variant(),
   Variant& v4 = Variant(),Variant& v5 = Variant(),Variant& v6 = Variant(),
   Variant& v7 = Variant(),Variant& v8 = Variant(),Variant& v9 = Variant());
 Variant OlePropertyGet(const String& name, Variant& v0 = Variant(),
   Variant& v1 = Variant(),Variant& v2 = Variant(),Variant& v3 = Variant(),
   Variant& v4 = Variant(),Variant& v5 = Variant(),Variant& v6 = Variant(),
   Variant& v7 = Variant(),Variant& v8 = Variant(),Variant& v9 = Variant());
 void OlePropertySet(const String& name, Variant& v0 = Variant(),
   Variant& v1 = Variant(),Variant& v2 = Variant(),Variant& v3 = Variant(),
   Variant& v4 = Variant(),Variant& v5 = Variant(),Variant& v6 = Variant(),
   Variant& v7 = Variant(),Variant& v8 = Variant(),Variant& v9 = Variant());
 // End of Alternate Syntax for Automation
```

TDateTime data type

See also Example Reference

The *TDateTime* class is a C++ class that implements the Delphi intrinsic type **TDateTime**.

Use the *TDateTime* data type to construct dates from the data types **double**, **int**, or *AnsiString*. The *TDateTime* class public interface allows you to do arithmetic and comparison operations on *TDateTime* types. You can also get information from a *TDateTime* type such as date, time, day of week, or a file’s date.

TDateTime

Syntax

```
class __declspec(delphireturn) TDateTime : public TDateTimeBase
```

Public constructors

```
// Used by TDateTime(const AnsiString& src)
enum TDateTimeFlag {Date, Time, DateTime};

static TDateTime __fastcall CurrentDate();
static TDateTime __fastcall CurrentTime();
static TDateTime __fastcall CurrentDateTime();
static TDateTime __fastcall FileDateToDateTime(int fileDate);

__fastcall TDateTime()                                  {Val = 0;}
__fastcall TDateTime(const TDateTimeBase& src)          {Val = src.Val;}
__fastcall TDateTime(const TDateTime& src)              {Val = src.Val;}
__fastcall TDateTime(const double src)                  {Val = src;}
__fastcall TDateTime(const int src)                     {Val = src;}
__fastcall TDateTime(const AnsiString& src, TDateTimeFlag flag = DateTime);
__fastcall TDateTime(unsigned short year, unsigned short month, unsigned
  short day);
__fastcall TDateTime(unsigned short hour, unsigned short min, unsigned short
  sec, unsigned short msec);
```

Operators

```
TDateTime& __fastcall operator =(const TDateTimeBase& rhs)
 {Val = rhs.Val;return *this;}
TDateTime& __fastcall operator =(const TDateTime& rhs)
 {Val = rhs.Val;  return *this;}
TDateTime& __fastcall operator =(const double rhs)
 {Val = rhs; return *this;}
TDateTime& __fastcall operator =(const int rhs)
 {Val = rhs; return *this;}

TDateTime& __fastcall operator +=(const TDateTimeBase& rhs)
 {Val+=rhs.Val;return *this;}
TDateTime& __fastcall operator +=(const TDateTime& rhs)
 {Val += rhs.Val; return *this;}
TDateTime& __fastcall operator +=(const double rhs)
 {Val += rhs; return *this;}
TDateTime& __fastcall operator +=(const int rhs)
 {Val += rhs; return *this;}

TDateTime& __fastcall operator -=(const TDateTimeBase& rhs)
 {Val-=rhs.Val;return *this;}
TDateTime& __fastcall operator -=(const TDateTime& rhs)
 {Val -= rhs.Val; return *this;}
TDateTime& __fastcall operator -=(const double rhs)
 {Val -= rhs; return *this;}
TDateTime& __fastcall operator -=(const int rhs)
 {Val -= rhs; return *this;}

TDateTime& operator ++() {Val++; return *this;}
TDateTime operator ++(int) {TDateTime tmp(*this); Val++; return tmp;}
TDateTime& operator --() {Val--; return *this;}
TDateTime operator --(int) {TDateTime tmp(*this); Val--; return tmp;}
TDateTime __fastcall operator +(const TDateTimeBase& rhs) const
 {return Val+rhs.Val;}
TDateTime __fastcall operator +(const TDateTime& rhs) const
 {return Val+rhs.Val;}
TDateTime __fastcall operator +(const double rhs) const {return Val+rhs;}
TDateTime __fastcall operator +(const int rhs) const {return Val+rhs;}

TDateTime __fastcall operator -(const TDateTimeBase& rhs) const
 {return Val-rhs.Val;}
TDateTime __fastcall operator -(const TDateTime& rhs) const
 {return Val-rhs.Val;}
TDateTime __fastcall operator -(const double rhs) const {return Val-rhs;}
TDateTime __fastcall operator -(const int rhs) const {return Val-rhs;}

// comparisons
bool __fastcall operator ==(const TDateTime& rhs) const
 {return Val == rhs.Val;}
bool __fastcall operator !=(const TDateTime& rhs) const
 {return Val != rhs.Val;}
bool __fastcall operator >(const TDateTime& rhs) const
 {return Val > rhs.Val;}
bool __fastcall operator <(const TDateTime& rhs) const
 {return Val < rhs.Val;}
bool __fastcall operator >=(const TDateTime& rhs) const
 {return Val >= rhs.Val;}
bool __fastcall operator <=(const TDateTime& rhs) const
 {return Val <= rhs.Val;}

__fastcall operator AnsiString() const;//<Date||Time||DateTime>String(smart)
AnsiString __fastcall FormatString(const AnsiString& format);
AnsiString __fastcall DateString() const;
AnsiString __fastcall TimeString() const;
AnsiString __fastcall DateTimeString() const;
__fastcall operator double() const;
__fastcall operator int() const;

int __fastcall DayOfWeek() const;
int __fastcall FileDate() const;
```

Example of base class referencing

```
template <class T> class B
{
  // class declarations
};
template <class T> class D : public B<T>
{
   // class declarations
};

template <class T> void func(B <T> *b)
{
   // function body
}
// This is illegal under ANSI C++: unresolved func( int )
// However, Borland C++ calls func( B<int> * ).
func( new D<int> );
```

Example of trivial conversions

```
template <class T> void func(const T)
{
   .
   .
   .
};
func(0);  // This is illegal under ANSI C++: unresolved func(int).
// However, Borland C++  allows func(const int) to be called.
```

Example of explicit template function

```
template<class T> T max(T a, T b) {
        return  (a > b) ? a : b;
};

// Declare explicit template function
int     max(int,int);

void    f(int i, char c)
{
        max(i, i);              // calls max(int ,int )
        max(c, c);              // calls max(char,char)
        max(i, c);              // calls max(int,int)
        max(c, i);              // calls max(int,int)
}
```

Class template definition

```
// An example for defining a template class.
template <class T> class Vector
{
   T *data;
   int size;
public:
   Vector(int);
   ~Vector( ) { delete[ ] data; }
   T& operator[ ] (int i) { return data[i]; }
};
// Note the syntax for out-of-line definitions.
template <class T> Vector<T>::Vector(int n)
{
   data = new T[n];
   size = n;
};

int main()
{
   Vector<int> x(5);   // Generate a vector to store five integers
   for (int i = 0; i < 5; ++i)
      x[i] = i;              //  Initialize the vector.
   return o;
}
```

Exportable/Importable Template Declarations

```
// In file EXPORTER.H
#include<iostream.h>
# if defined (BUILD_DLL_EXPORTS)
#     define DECLSPEC __export
# elif defined (USING_DLL_IMPORTS)
#     define DECLSPEC __import
# endif

//////////////////////////////////////////////////
// Receive CLASS DEFINITIONS
template <class T> class Receive
{
  T value;
public:
  Receive(const T val) : value(val){}
   T display();
};

template<class T> T Receive<T>::display()
{
  return value;
}

// TEMPLATE FUNCTION DEFINITION
template <class T>
T another_min(T a, T b) { return a < b ? a : b;}

#if (defined (BUILD_DLL_EXPORTS) ||  defined(USING_DLL_IMPORTS) )
////// INSTANTIATED TEMPLATE CLASSES /////
template class DECLSPEC Receive<double>;
template class DECLSPEC Receive<int>;
template class DECLSPEC Receive<char>;

////// INSTANTIATED TEMPLATE FUNCTIONS /////
template int DECLSPEC another_min<int>(int, int);
template double DECLSPEC another_min<double>(double, double);
#endif
```

Compiling Exportable Templates

```
// In file DLL_SRC.CPP.
// GENERATE CODE FOR EXPORTABLE CLASSES AND FUNCTIONS.
// TO COMPILE THIS FILE, USE BCC32 -tWD -DBUILD_DLL_EXPORTS DLL_SRC.CPP
#define STRICT
#include <windows.h>
#include "exporter.h"

BOOL WINAPI DllEntryPoint(HINSTANCE hinstdll,
                          DWORD fdwReason, LPVOID lpvReserved)
{
  return 1;
}
```

Using Import Templates

Program Output

```
// Before you compile this file you need to create the dynamic link library.
// You can use the command IMPLIB DLL_SRC.LIB DLL_SRC.DLL

// TO COMPILE THIS FILE, USE BCC32 -DUSING_DLL_IMPORTS MAIN DLL_SRC.LIB
#include <iostream.h>
#include "exporter.h"

int main () {
   int small = 5;
   int big = 10;
   double smalld = 1.2;
   double bigd = 12.3;

   // No new code is generated for these objects.
   Receive <double> Test_d(0.01);
   Receive <int> Test_i(5);

  // Generate code in MAIN.OBJ for this object.
  Receive <float> Test_f(3.14);
   cout << "Test_d.display() = " <<  Test_d.display() << endl;
   cout << "Test_i.display() = " <<  Test_i.display() << endl;

   cout << "min(5, 10): " << another_min(small, big) << endl;
   cout << "min(12.3, 1.2): " << another_min(bigd, smalld)<<endl;
   cout << "Test_f.display() = " <<  Test_f.display() << endl;

   return 0;
}
```

Program Output

```
Test_d.display() = 0.01
Test_i.display() = 5
min(5, 10): 5
min(12.3, 1.2): 1.2
Test_f.display() = 3.14
```

// Example of the new and delete Operators

```
// ALLOCATE A TWO-DIMENSIONAL SPACE, INITIALIZE, AND DELETE IT.
#include <except.h>
#include <iostream.h>

void display(long double **);
void de_allocate(long double **);

int m = 3;                              // THE NUMBER OF ROWS.
int n = 5;                              // THE NUMBER OF COLUMNS.

int main(void) {
   long double **data;

   try {                                // TEST FOR EXCEPTIONS.
      data = new long double*[m];       // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < m; j++)
          data[j] = new long double[n]; // STEP 2: SET UP THE COLUMNS
      }
   catch (xalloc) {  // ENTER THIS BLOCK ONLY IF xalloc IS THROWN.
      // YOU COULD REQUEST OTHER ACTIONS BEFORE TERMINATING
      cout << "Could not allocate. Bye ...";
      exit(-1);
      }

   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
          data[i][j] = i + j;           // ARBITRARY INITIALIZATION

   display(data);
   de_allocate(data);
   return 0;
   }

void display(long double **data) {
   for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
             cout << data[i][j] << " ";
       cout << "\n" << endl;
       }
   }

void de_allocate(long double **data) {
   for (int i = 0; i < m;  i++)
       delete[] data[i];                // STEP 1: DELETE THE COLUMNS

   delete[] data;                       // STEP 2: DELETE THE ROWS
   }
```

operator new placement syntax example

```
// An example of the placement syntax for operator new()
#include <iostream.h>

class Alpha {
  union {
    char  ch;
    char  buf[10];
    };
public:
   Alpha(char c = '\0') : ch(c) {
      cout << "character constructor" << endl;
      }
   Alpha(char *s) {
      cout << "string constructor" << endl;
      strcpy(buf,s);
      }

   ~Alpha( ) { cout << "Alpha::~Alpha() " << endl; }

   void * operator new(size_t, void * buf) {
      return buf;
      }
};

void main() {
   char *str = new char[sizeof(Alpha)];

   // Place 'X' at start of str.
   Alpha* ptr = new(str) Alpha('X');
   cout << "str[0] = " << str[0] << endl;

   // Explicit call of the destructor
   ptr -> Alpha::~Alpha();

   // Place a string in str buffer.
   ptr = new(str) Alpha("my string");
   cout << "\n str = " << str << endl;

   // Explicit call of the destructor
   ptr -> Alpha::~Alpha();
   delete[] str;
   }
```

Output

```
character constructor
str[0] = X
Alpha::~Alpha()
string constructor

 str = my string
Alpha::~Alpha()
```

Example of Overloading the new and delete Operators

```
#include <stdlib.h>

class X {
   .
   .
   .
public:
   void* operator new(size_t size) { return newalloc(size);}
   void operator delete(void* p) { newfree(p); }
   X() { /* initialize here */ }
   X(char ch) { /* and here */ }

   ~X() { /* clean up here */ }
   .
   .
   .
};
```

Example

```
// The MakeStr function takes an "array of const"

AnsiString MakeStr(const TVarRec* args, int argHigh)
{
  String tmp;

  for (int i = 0; i <= argHigh; i++)
  switch (args[i].VType)
  {
    case vtInteger:
      tmp += ::IntToStr(args[i].VInteger);
      break;
    case vtBoolean:
      tmp += args[i].VBoolean? "T": "F";
      break;
    case vtChar:
      tmp += String(&args[i].VChar, 1);
      break;
    case vtExtended:
      tmp +=  ::FloatToStr(*args[i].VExtended);
      break;
    case vtString:
      tmp += *args[i].VString;
      break;
    case vtPChar:
      tmp += String(args[i].VPChar);
      break;
    case vtObject:
      tmp += "-TObject-"; //classname not available args[i].VObject-
  >ClassName();
      break;
    case vtClass:
      tmp += "-TClass-"; //classname not available args[i].VClass-
  >ClassName();
      break;
    case vtAnsiString:
      tmp += String(reinterpret_cast<char*>(args[i].VAnsiString));
      break;
    case vtCurrency:
      tmp += ::CurrToStr(*args[i].VCurrency);
      break;
    case vtVariant:
      tmp += *args[i].VVariant;
      break;
    default:
      tmp += "-Other-";
  };
  return tmp;
}

// MakeStr is called with OPENARRAY(TVarRec, (...))
  ::MessageBox(0,
    MakeStr(OPENARRAY(TVarRec,(1, true, 'a', (long double)3.14159,
    ShortString("shstr"), "hello", *this, __classid(TForm1), String("str"),
    Currency(1.23), Variant("var")))).c_str(), PChar("MakeStr"), MB_OK);
```

```
// The resulting message box contains:
//  1Ta3.14159shstrhello-TObject--TClass-str1.23var
```