made with doc-o-matic®

Delphi C++ Reference Guide

Copyright© 2007 CodeGear™. All Rights Reserved.

Table of Contents

1 Reference

1.1 C++ Reference

This section contains reference topics for the C++ library in RAD Studio.

1.1.1 C++ Language Guide

This sections contains C++ language topics.

1.1.1.1 Lexical Elements

This section contains Lexical Element topics.

1.1.1.1.1 Lexical Elements

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

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

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

A C++ program starts as a sequence of ASCII characters representing the source code, created using a suitable text editor (such as the IDE's editor). The basic program unit in C++ is a source file (usually designated by a ".c", or ".cpp" in its name), and all of the header files and other source files included with the **#include** preprocessor directive. Source files are usually designated by a ".c" or ".cpp" in the name, while header files are usually designated with a ".h" or ".hpp".

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

See Also

Whitespace (see page 2)

Tokens (see page 5)

1.1.1.1.2 Whitespace Overview

This section contains Whitespace Overview topics.

1.1.1.1.2.1 Whitespace

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

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

and

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

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

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

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

```
char name[] = "CodeGear Corporation";
```

parses to seven tokens, including the single literal-string token "CodeGear Corporation"

**Line splicing with **

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

```
"CodeGear \
Corporation"
```

is parsed as "CodeGear Corporationl" (see String constants for more information).

See Also

Lexical Elements (see page 2)

Tokens (see page 5)

1.1.1.1.2.2 Comments

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

There are two ways to delineate comments: the C method and the C++ method. The compiler supports both methods, and provides an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

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

C comments

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

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

```
#define VAR(i,j)  (i/**/j)    /* won't work */
#define VAR(i,j)  (i##j)      /* OK */
```

```
#define VAR(i,j)  (i ## j)    /* Also OK */
```

The compiler parses the declaration,

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

as these three tokens:

```
int   i;
```

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

C++ comments

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

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

You can also use // to create comments in C code. This feature is specific to the CodeGear C++ compiler and is generally not portable.

Nested comments

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

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

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

```
i ; */
```

which would generate a syntax error.

To allow nested comments, check Project|Options|Advanced Compiler|Source|Nested Comments.

Delimiters and whitespace

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

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

parses as int i = j +m; not as

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

as expected under the C convention. The more legible

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

avoids this problem.

See Also

Lexical Elements (see page 2)

Whitespace (see page 2)

Tokens (see page 5)

1.1.1.1.3 Tokens Overview

This section contains Token topics.

1.1.1.1.3.1 Tokens

Tokens are word-like units recognized by a language. The compiler recognizes six classes of tokens.

Here is the formal definition of a token:

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

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

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

See Also

Lexical Elements (see page 2)

Whitespace (see page 2)

1.1.1.1.3.2 Keywords Overview

This section contains Keyword topics.

1.1.1.1.3.2.1 Keywords

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

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

Note: Note that the keywords __**try** and **try** are an exception to the discussion above. The keyword **try** is required to match the **catch** keyword in the C++ exception-handling mechanism. **try** cannot be substituted by __**try.** The keyword __**try** can only be used to match the __**except** or __**finally** keywords. See the discussions on C++ exception handling and C-based structured exceptions under Win32 for more information.

Please see the Help table of contents for a complete categorical and alphabetical listing of keywords.

1.1.1.1.3.2.2 C++-Specific Keywords

A number of keywords are specific to C++ and are not available if you are writing a program in C only. Please see the Help table of contents for a complete categorical and alphabetical listing of these and other keywords.

1.1.1.1.3.2.3 Table Of CodeGear C++ Register Pseudovariables

This table lists all pseudovariables.

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

All but the _FLAGS and _EFLAGS register pseudovariables 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 flags registers contain information about the state of the 80x86 and the results of recent instructions.

1.1.1.1.3.2.4 Keyword Extensions

CodeGear C++ provides a number of keywords that are not part of the ANSI Standard.

Please see the Help table of contents for a complete categorical and alphabetical listing of Library Routines.

1.1.1.1.3.3 Identifiers Overview

1

This section contains Identifier topics.

1.1.1.1.3.3.1 Identifiers

Here is the formal definition of an identifier:

identifier:

- nondigit
- identifier nondigit
- identifier digit

nondigit: one of

- a b c d e f g h i j k l m n o p q r s t u v w x y z _
- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

- 0 1 2 3 4
- ef5 6 7 8 9

Naming and length restrictions

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

- The first character must be a letter or an underscore.
- By default, the compiler recognizes only the first 250 characters as significant. The number of significant characters can be

reduced by menu and command-line options, but not increased. To change the significant character length, use the spin control in Project|Options|Advanced Compiler|Source|Identifier Length.

Case sensitivity

Identifiers in C and C++ are case sensitive, so that Sum, sum and suM are distinct identifiers.

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

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

Uniqueness and scope

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

1.1.1.1.3.4 Constants Overview

This section contains Constant topics.

1.1.1.1.3.4.1 Constants

Constants are tokens representing fixed numeric or character values.

The compiler supports four classes of constants: integer, floating point, character (including strings), and enumeration.

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

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

Constants: Formal Definitions

constant: floating-constant integer-constant numeration-constant character-constant	nonzero-digit: one of 1 2 3 4 5 6 7 8 9
floating-constant: fractional-constant <exponent-part> <floating-suffix> digit-sequence exponent-part <floating-suffix>	octal-digit: one of 0 1 2 3 4 5 6 7
fractional-constant: <digit-sequence> **.** digit-sequence digit-sequence **.** a b c d e f	hexadecimal-digit: one of 0 1 2 3 4 5 6 7 8 9 A B C D E F
exponent-part: e <sign> digit-sequence E <sign> digit-sequence	integer-suffix: unsigned-suffix <long-suffix> long-suffix <unsigned-suffix>
sign: one of • -	unsigned-suffix: one of u U

digit-sequence: digit digit-sequence digit	long-suffix: one of l L
floating-suffix: one of f l F L	enumeration-constant: identifier
integer-constant: decimal-constant <integer-suffix> octal-constant <integer-suffix> hexadecimal-constant <integer-suffix>	character-constant c-char-sequence
decimal-constant: nonzero-digit decimal-constant digit	c-char-sequence: c-char c-char-sequence c-char
octal-constant: 0 octal-constant octal-digit	c-char: Any character in the source character set except the single-quote ('), backslash (\), or newline character escape-sequence.
hexadecimal-constant: 0 x hexadecimal-digit 0 X hexadecimal-digi hexadecimal-constant hexadecimal-digit	escape-sequence: one of the following \" \' \? \\ t \a \b \f \n \o \oo \ooo \r \t \v \Xh... \xh...

See Also

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.2 Integer Constants

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

Decimal

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

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

Octal

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

Hexadecimal

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

long and unsigned suffixes

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

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

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

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

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

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

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

See Also

Constants (see page 7)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

__int8 (see page 10)

1.1.1.1.3.4.3 __int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types

Category

Keyword extensions

Description

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

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.4 Integer Constant Without L Or U

Decimal constants

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

Octal constants

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

Hexadecimal constants

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

__int8 (see page 10)

1.1.1.1.3.4.5 Floating Point Constants

A floating-point constant consists of:

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

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

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

Here are some examples:

Constant	Value
23.45e6	23.45 (10^6
.0	0
0.	0.0
1.	1.0
-1.23	-1.23
2e-5	2.0 (10^-5
3E+10	3.0 (10^10
.09E34	0.09 (10^34

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

Floating-point constant sizes and ranges

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.6 Character Constants Overview

This section contains Character Constant topics.

1.1.1.1.3.4.6.1 Character Constants

A character constant is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-character constants have data type **int**. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

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

- Three char types (see page 14)
- Escape sequences (see page 14)
- Wide-character and multi-character constants (see page 16)

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.6.2 The Three Char Types

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

The three character types, **char**, **signed char**, and **unsigned char,** require an 8-bit (one byte) storage. By default, the compiler treats character declarations as signed. Use the –K compiler option to treat character declarations as unsigned. The behavior of C programs is unaffected by the distinction between the three character types.

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

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

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

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.6.3 Escape Sequences

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

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

octal or hexadecimal escape sequence marks the end of the sequence.

Take this example.

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

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

To avoid such problems, rewrite your code like this:

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

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

The following table shows the available escape sequences.

Escape sequences

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

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.6.4 Wide-character And Multi-character Constants

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

```
wchar_t ch = L'A';
```

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

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

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

Multi-character constants

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.7 String Constants

This section contains String Constant topics.

1.1.1.1.3.4.7.1 String Constants

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of- **const 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 the compiler "   " will\nconcatenate very long strings for
you"   " automatically, \nresulting in nicer" " looking programs.";
   printf(p);
   return(0);
}
```

The output of the program is

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

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

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.8 Enumeration Constants

This section contains Enumeration Constant topics.

1.1.1.1.3.4.8.1 Enumeration Constants

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

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

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

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

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

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

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

the constants are set as follows:

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

The constant values need not be unique:

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

1.1.1.1.3.4.9 Constants and Internal Representation

This section contains Constants and Internal Representation topics.

1.1.1.1.3.4.9.1 Constants And Internal Representation

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

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

32-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 <= X <= 255	Small numbers and full PC character set
char	8	-128 <= X <= 127	Very small numbers and ASCII characters
short int	16	-32,768 <= X <= 32,767	Counting, small numbers, loop control
unsigned int	32	0 <= X <= 4,294,967,295	Large numbers and loops
int	32	-2,147,483,648 <= X <= 2,147,483,647	Counting, small numbers, loop control
unsigned long	32	0 <= X <= 4,294,967,295	Astronomical distances
enum	32	-2,147,483,648 <= X <= 2,147,483,647	Ordered sets of values
long	32	-2,147,483,648 <= X <= 2,147,483,647	Large numbers, populations
float	32	1.18 (10^-38 < |X| < 3.40 (10^38	Scientific (7-digit) precision)
double	64	2.23 (10^-308 < |X| < 1.79 (10^308	Scientific (15-digit precision)
long double	80	3.37 (10^-4932 < |X| < 1.18 (10^4932	Financial (18-digit precision)

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Internal Representation Of Numerical Types (see page 20)

Constant Expressions (see page 21)

__int8 (see page 10)

The Three Char Types (see page 14)

Internal Representation Of Numerical Types (see page 20)

1.1.1.1.3.4.10 Internal Representation of Numerical Types

This section contains Internal Representation of Numerical Type topics.

1.1.1.1.3.4.10.1 Internal Representation Of Numerical Types

32-bit integers

Floating-point types, always

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

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Constant Expressions (see page 21)

1.1.1.1.3.4.11 Constant Expressions

This section contains Constant Expression topics.

1.1.1.1.3.4.11.1 Constant Expressions

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

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

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

- Assignment
- Comma
- Decrement
- Function call
- Increment

See Also

Constants (see page 7)

Integer Constants (see page 8)

Integer Constant Without L Or U (see page 10)

Floating Point Constants (see page 11)

Character Constants (see page 13)

The Three Char Types (see page 14)

Escape Sequences (see page 14)

Wide-character And Multi-character Constants (see page 16)

String Constants (see page 17)

Enumeration Constants (see page 18)

Constants And Internal Representation (see page 19)

Internal Representation Of Numerical Types (see page 20)

1.1.1.1.3.5 Punctuators Overview

This section contains Punctuator topics.

1.1.1.1.3.5.1 Punctuators

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

- [ ]
- ( )
- { }

- ,
- ;
- :
- ...
- =
- #

Most of these punctuators also function as operators.

Brackets

Open and close brackets indicate single and multidimensional array subscripts:

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

Parentheses

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

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

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

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

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

Braces

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

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

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

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

Comma

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

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

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them. Note that (exp1, exp2) evaluates both but is equal to the second:

```
func(i, j);                          /* call func with two args */
```

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

Semicolon

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

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

Semicolons are often used to create an empty statement:

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

Colon

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

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

Labels are discussed in Labeled statements.

Ellipsis

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

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

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

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

Asterisk (pointer declaration)

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

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

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

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

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

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer)

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

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

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

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

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

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

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

Pound sign (preprocessor directive)

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

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

1.1.1.2 Language Structure

This section contains Language Structure topics.

1.1.1.2.1 Language Structure

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

See Also

Declarations (see page 24)

Declaration Syntax (see page 31)

Pointers (see page 51)

Arrays (see page 54)

Functions (see page 55)

Structures (see page 58)

Unions (see page 64)

Enumerations (see page 66)

Expressions (see page 67)

Statements (see page 85)

1.1.1.2.2 Declarations

This section contains Declaration topics.

1.1.1.2.2.1 Declarations

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

See Also

Objects (see page 25)

Storage Classes And Types (see page 26)

Scope (see page 26)

Visibility (see page 27)

Duration (see page 28)

Linkage (see page 30)

1.1.1.2.2.2 Objects

This section contains Object topics.

1.1.1.2.2.2.1 Objects

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

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

Borland's C++ compiler supports all standard data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in memory.

Objects and declarations

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

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

lvalues

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

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

rvalues

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

See Also

Declarations (see page 24)

Storage Classes And Types (see page 26)

Scope (see page 26)

Visibility (see page 27)

Duration (see page 28)

Linkage (see page 30)

1.1.1.2.2.3 Storage Classes And Types

This section contains Storage Classes and Type topics.

1.1.1.2.2.3.1 Storage Classes And Types

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

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

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

See Also

Declarations (see page 24)

Objects (see page 25)

Scope (see page 26)

Visibility (see page 27)

Duration (see page 28)

Linkage (see page 30)

1.1.1.2.2.4 Scope

This section contains Scope topics.

1.1.1.2.2.4.1 Scope

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

- **Block**. The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the enclosing block). Parameter declarations with a function definition

also have block scope, limited to the scope of the block that defines the function.

- **Function**. The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing label_name: followed by a statement. Label names must be unique within a function.
- **Function prototype**. Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.
- **File**. File scope identifiers, also known as globals, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.
- **Class** (C++). A class is a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see Classes.
- **Condition** (C++). Declarations in conditions are supported. Variables can be declared within the expression of **if**, **while**, and **switch** statements. The scope of the variable is that of the statement. In the case of an **if** statement, the variable is also in scope for the **else** block.
- **namespace** (C++). A namespace is a logical grouping of program entities (e.g. identifiers, classes, and functions). Namespaces are open, that is, they can span multiple compilation units. You can think of a namespace as introducing a named scope, similar in many ways to a class in C++. See the help for the namespace keyword for more inforrmation on how to declare and use namespaces.

Name spaces

Name space is the scope within which an identifier must be unique. Note that a C++ namespace extends this concept by allowing you to give the scope a name. In addition to the named-scoping capability of C++, the C programming language uses four distinct classes of identifiers:

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

See Also

Declarations (see page 24)

Objects (see page 25)

Storage Classes And Types (see page 26)

Visibility (see page 27)

Duration (see page 28)

Linkage (see page 30)

1.1.1.2.2.5 Visibility

This section contains Visibility topics.

1.1.1.2.2.5.1 Visibility

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

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the

appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

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

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

See Also

Declarations (see page 24)

Objects (see page 25)

Storage Classes And Types (see page 26)

Scope (see page 26)

Duration (see page 28)

Linkage (see page 30)

1.1.1.2.2.6 Duration

This section contains Duration topics.

1.1.1.2.2.6.1 Duration

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **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. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

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

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

Local

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

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

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

Dynamic

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

See Also

Declarations (see page 24)

Objects (see page 25)

Storage Classes And Types (see page 26)

Scope (see page 26)

Visibility (see page 27)

Linkage (see page 30)

1.1.1.2.2.6.2 static

Category

Storage class specifiers

Syntax

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

Description

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

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

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

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

1.1.1.2.2.7 Translation Units

This section contains Translation Unit topics.

1.1.1.2.2.7.1 Translation Units

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

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

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

See Also

Declarations (see page 24)

Objects (see page 25)

Storage Classes And Types (see page 26)

Scope (see page 26)

Visibility (see page 27)

Duration (see page 28)

Linkage (see page 30)

1.1.1.2.2.8 Linkage

This section contains Linkage topics.

1.1.1.2.2.8.1 Linkage

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

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

External and internal linkage rules

Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.

For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.

If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.

If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.

If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

Identifiers with no linkage attribute:

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

Name mangling

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

functions from the other module. This situation typically arises when linking with libraries or .obj files compiled with a C compiler

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

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

This declaration tells the compiler not to mangle references to the function Cfunc.

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

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

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

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

Note: extern “C” cannot be used with class identifiers.

See Also

Declarations (see page 24)

Objects (see page 25)

Storage Classes And Types (see page 26)

Scope (see page 26)

Visibility (see page 27)

Duration (see page 28)

1.1.1.2.3 Declaration Syntax

This section contains Declaration Syntax topics.

1.1.1.2.3.1 Declaration Syntax

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

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

See Also

Tentative Definitions (see page 32)

Possible Declarations (see page 32)

External Declarations And Definitions (see page 35)

Type Categories (see page 38)

The Fundamental Types (see page 39)

Initialization (see page 42)

Declarations And Declarators (see page 43)

Variable Modifiers (see page 45)

Function Modifiers (see page 50)

1.1.1.2.3.2 Tentative Definitions

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

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

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

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

See Also

Declaration Syntax (see page 31)

Possible Declarations (see page 32)

External Declarations And Definitions (see page 35)

Type Categories (see page 38)

The Fundamental Types (see page 39)

Initialization (see page 42)

Declarations And Declarators (see page 43)

Variable Modifiers (see page 45)

Function Modifiers (see page 50)

1.1.1.2.3.3 Possible Declarations

The range of objects that can be declared includes

- Variables
- Functions
- Classes and class members (C++)
- Types
- Structure, union, and enumeration tags

- Structure members
- Union members
- Arrays of other types
- Enumeration constants
- Statement labels
- Preprocessor macros

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

In CodeGear 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.

CodeGear C++ declaration syntax

declaration:	elaborated-type-specifier:
<decl-specifiers> <declarator-list>;	class-key identifier
asm-declaration	class-key class-name
function-declaration	**enum** enum-name
linkage-specification	class-key: (C++ specific)
decl-specifier:	**class**
storage-class-specifier	**struct**
type-specifier	**union**
function-specifier	enum-specifier:
friend (C++ specific)	**enum** <identifier> { <enum-list> }
typedef	enum-list:
decl-specifiers:	enumerator
<decl-specifiers> decl-specifier	enumerator-list , enumerator
storage-class-specifier:	enumerator:
auto	identifier
register	identifier = constant-expression
static	constant-expression:
extern	conditional-expression
function-specifier: (C++ specific)	linkage-specification: (C++ specific)
inline	**extern** string { <declaration-list> }
virtual	**extern** string declaration
simple-type-name:	type-specifier:
class-name	simple-type-name
typedef-name	class-specifier
boolean	

char	enum-specifier
short	elaborated-type-specifier
int	**const**
__int8	
__int16	
__int32	
__int64	
long	**volatile**
signed	declaration-list:
unsigned	declaration
float	declaration-list ; declaration
double	
void	
declarator-list:	type-name:
init-declarator	type-specifier <abstract-declarator>
declarator-list , init-declarator	abstract-declarator:
init-declarator:	pointer-operator <abstract-declarator>
declarator <initializer>	<abstract-declarator> (argument-declaration-list)
declarator:	<cv-qualifier-list>
dname	<abstract-declarator> [<constant-expression>]
modifier-list	(abstract-declarator)
pointer-operator declarator	argument-declaration-list:
declarator (parameter-declaration-list)	<arg-declaration-list>
<cv-qualifier-list >	arg-declaration-list , ...
(The <cv-qualifier-list > is for C++ only.)	<arg-declaration-list> **...** (C++ specific)
declarator [<constant-expression>]	arg-declaration-list:
(declarator)	argument-declaration
modifier-list:	arg-declaration-list , argument-declaration
modifier	argument-declaration:
modifier-list modifier	decl-specifiers declarator
modifier:	decl-specifiers declarator = expression
__cdecl	(C++ specific)
__pascal	decl-specifiers <abstract-declarator>
__stdcall	decl-specifiers <abstract-declarator> **=** expression

__fastcall	(C++ specific)
function-definition:	
function-body:	
pointer-operator:	compound-statement
• <cv-qualifier-list>	initializer:
& <cv-qualifier-list> (C++ specific)	**=** expression
class-name :: * <cv-qualifier-list>	= { initializer-list }
(C++ specific)	(expression-list) (C++ specific)
cv-qualifier-list:	initializer-list:
cv-qualifier <cv-qualifier-list>	expression
cv-qualifier	initializer-list , expression
const	{ initializer-list <,> }
volatile	
dname:	
name	
class-name (C++ specific)	
~ class-name (C++ specific)	
type-defined-name	

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

External Declarations And Definitions (see page 35)

Type Categories (see page 38)

The Fundamental Types (see page 39)

Initialization (see page 42)

Declarations And Declarators (see page 43)

Variable Modifiers (see page 45)

Function Modifiers (see page 50)

1.1.1.2.3.4 External Declarations And Definitions

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

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

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

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

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

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

class-specifier: base-specifier:

class-head { <member-list> } : base-list

class-head: base-list:

class-key <identifier> <base-specifier> base-specifier

class-key class-name <base-specifier> base-list , base-specifier

member-list: base-specifier:

member-declaration <member-list> class-name

access-specifier : <member-list> **virtual** <access-specifier> class-name

member-declaration: access-specifier <**virtual**> class-name

<decl-specifiers> <member-declarator-list> ; access-specifier:

function-definition <;> **private**

qualified-name ; **protected**

member-declarator-list: **public**

member-declarator conversion-function-name:

member-declarator-list, member-declarator **operator** conversion-type-name

member-declarator: conversion-type-name:

declarator <pure-specifier> type-specifiers <pointer-operator>

<identifier> : constant-expression constructor-initializer:

pure-specifier: **:** member-initializer-list

= 0

member-initializer-list: operator-name: one of

member-initializer **new delete sizeof typeid**

member-initializer , member-initializer-list **+ - * / % ^**

member-initializer: **& | ~ ! = <>**

class name (<argument-list>)

+= -= =* /= %= ^=

identifier (<argument-list>) **&= |= << >> >>= <<=**

operator-function-name: **== != <= >= && ||**

operator operator-name **++ __ , ->* -> ()**

[ ] .*

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

Possible Declarations (see page 32)

Type Categories (see page 38)

The Fundamental Types (see page 39)

Initialization (see page 42)

Declarations And Declarators (see page 43)

Variable Modifiers (see page 45)

Function Modifiers (see page 50)

1.1.1.2.3.5 Type Specifiers

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

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

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

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

The type specifier keywords by the CodeGear C++ compiler are:

char wchar_t	float	signed
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.

1.1.1.2.3.6 Type Categories

This section contains Type Category topics.

1.1.1.2.3.6.1 Type Categories

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

Aggregate

Array

struct

union

class (C++ only)

Function

Scalar

Arithmetic

Enumeration

Pointer

Reference (C++ only)

void

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

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

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

Declaring types

Declaration	Description
type t;	An object of type **type**
type array[10];	Ten **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.

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

Possible Declarations (see page 32)

External Declarations And Definitions (see page 35)

1.1.1.2.3.6.2 void

Category

Special types

Syntax

```
void identifier
```

Description

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

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

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

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

Void Pointers

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

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

1.1.1.2.3.7 The Fundamental Types

This section contains Fundamental Type topics.

1.1.1.2.3.7.1 The Fundamental Types

The fundamental type specifiers are built from the following keywords:

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

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

Integral types

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

Integral types

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

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

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

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

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

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

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

Floating-point types

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

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

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

Standard arithmetic conversions

When you use an arithmetic expression, such as a + b, where a and b are different arithmetic types, The compiler performs

certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps the compiler uses to convert the operands in an arithmetic expression:

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

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

Methods used in standard arithmetic conversions

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

Special char, int, and enum conversions

Note: The conversions discussed in this section are specific to the Borland C++ compiler.

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

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

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

Possible Declarations (see page 32)

External Declarations And Definitions (see page 35)

Type Categories (see page 38)

1.1.1.2.3.8 Initialization

This section contains Initialization topics.

1.1.1.2.3.8.1 Initialization

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

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

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

Syntax for initializers

```
initializer
        = expression
        = {initializer-list} <,>}
        (expression list)
initializer-list
        expression
        initializer-list, expression
        {initializer-list} <,>}
```

Rules governing initializers

The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.

The item to be initialized must be an object (for example, an array).

For C (not required for C++), all expressions must be constants if they appear in one of these places:

In an initializer for an object that has static duration.

In an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed).

- If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
- If a brace-enclosed list has fewer initializers than members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

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

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

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

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or

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

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

The following rules initialize character arrays and wide character arrays:

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

*

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

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

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

* Here is an example of a structure initialization:

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

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

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

Possible Declarations (see page 32)

External Declarations And Definitions (see page 35)

Type Categories (see page 38)

The Fundamental Types (see page 39)

Declarations And Declarators (see page 43)

Variable Modifiers (see page 45)

Function Modifiers (see page 50)

1.1.1.2.3.9 Declarations And Declarators

This section contains Declarations And Declarator topics.

1.1.1.2.3.9.1 Declarations And Declarators

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

Simple declarations of variable identifiers have the following pattern:

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

where var1, var2,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of

type data-type. For example,

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

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

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

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

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

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

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

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

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

Declaration syntax examples

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

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

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

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

1.1.1.2.3.10 Use Of Storage Class Specifiers

This section contains Use Of Storage Class Specifier topics.

1.1.1.2.3.10.1 Storage Class Specifiers

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

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

The storage class specifiers in C++ are:

- auto register
- __declspec static
- extern typedef
- mutable

1.1.1.2.3.11 Variable Modifiers

This section contains Variable Modifier topics.

1.1.1.2.3.11.1 Variable Modifiers

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

Mixed-language calling conventions

You can call routines written in other languages, and vice versa. When you mix languages, you have to deal with two important issues: identifiers and parameter passing.

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

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

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

Calling conventions

Modifier	Push parameters	Pop parameters	Name change (only in C)
__cdecl1	Right to left	Caller	'_' prepended
__fastcall	Left to right	Callee	'@' prepended
__pascal	Left to right	Callee	Uppercase
__stdcall	Right to left	Callee	No change

1. This is the default.

Note: Note: __fastcall and __stdcall are always name mangled in C++. See the description of the -VC option.

See Also

Declaration Syntax (see page 31)

Tentative Definitions (see page 32)

Possible Declarations (see page 32)

External Declarations And Definitions (see page 35)

Type Categories (see page 38)

The Fundamental Types (see page 39)

Initialization (see page 42)

Declarations And Declarators (see page 43)

Variable Modifiers

Function Modifiers (see page 50)

1.1.1.2.3.11.2 const

Category

Modifiers

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;

// When used by itself, const is equivalent to int.
const  maxint  = 12345;

// A constant pointer
char *const str1 = "Hello, world";
```

```
// A pointer to a constant character string.
char const *str2 = "CodeGear Corporation";
```

Given these declarations, the following statements are illegal.

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

Using the const Keyword in C++ Programs

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

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

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

```
*(int *)&my_age = 35;
```

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

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

printf is prevented from modifying the format string.

See Also

Mutable (see page 172)

1.1.1.2.3.11.3 volatile

Category

Modifiers

Syntax

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

Description

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

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

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

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

See Also

Const (see page 46)

1.1.1.2.3.12 Mixed-Language Calling Conventions

This section contains Mixed-Language Calling Convention topics.

1.1.1.2.3.12.1 cdecl, _cdecl, __cdecl

Category

Modifiers, Keyword extensions

Syntax

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

Description

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

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

1.1.1.2.3.12.2 pascal, _pascal, __pascal

Category

Modifiers, Keyword extensions

Syntax

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

Description

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

In addition, **pascal** declares Delphi language-style parameter-passing conventions when applied to a function header (parameters pushed left to right; the called function cleans up the stack).

In C++ programs, functions declared with the **pascal** modifier are still mangled.

1.1.1.2.3.12.3 _stdcall, __stdcall

Category

Modifiers, Keyword extensions

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

1.1.1.2.3.12.4 _fastcall, __fastcall

Category

Modifiers, Keyword extensions

Syntax

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

Description

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

Note: All VCL class member functions and form class member functions must use the __**fastcall** convention.

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

Functions declared using _**cdecl** or _**pascal** cannot have the _**fastcall** modifier because they use the stack to pass parameters. Likewise, the _**fastcall** modifier cannot be used together with _export.

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

For Microsoft C++ style __**fastcall** implementation, see __msfastcall and __msreturn.

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

1.1.1.2.3.13 Multithread Variables

This section contains Multithread Variable topics.

1.1.1.2.3.13.1 __thread, Multithread Variables

Category

Keyword extensions

Description

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

The syntax is Type __**thread** variable__name. For example

```
int __thread x;
```

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

1.1.1.2.3.14 Function Modifiers

This section contains Function Modifier topics.

1.1.1.2.3.14.1 Function Modifiers

This section presents descriptions of the function modifiers available with the CodeGear C++ compiler.

You can use the __**declspec(dllexport),** and __**declspec(dllimport)**and **__**saveregs modifiers to modify functions.

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

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

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

CodeGear C++ 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 in C.
__pascal	Functions	Forces Pascal argument-passing convention. Affects linker and link-time names.
__pascal	Variables	Forces global identifier case-insensitivity with no leading underscores in C.
__import	Functions/classes	Tells the compiler which functions or classes to import.
__export	Functions/classes	Tells the compiler which functions or classes to export.
__declspec(dllimport)	Functions/classes	Tells the compiler which functions or classes to import. This is the preferred method.
__declspec(dllexport)	Functions/classes	Tells the compiler which functions or classes to export. This is the preferred method.
__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.

See Also

Declaration Syntax (see page 31)

1.1.1.2.4 Pointers

This section contains Pointer topics.

1.1.1.2.4.1 Pointers

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

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

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

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

See Also

1.1.1.2.4.2 Pointers To Objects

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

1.1.1.2.4.3 Pointers To Functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called.

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

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

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

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

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

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

1.1.1.2.4.4 Pointer Declarations

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

Warning: You need to initialize pointers before using them.

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

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

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

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

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

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

```
void *vptr;
```

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

1.1.1.2.4.5 Pointer Constants

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

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

```
const int * const cpc = &ci; // cpc is a constant pointer to a
                             // constant int
```

The following assignments are legal:

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

The following assignments are illegal:

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

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

1.1.1.2.4.6 Pointer Arithmetic

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

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

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

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

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

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

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

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

1.1.1.2.4.7 Pointer Conversions

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

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

More generally, the cast (**type***) will convert a pointer to type "pointer to **type**."

See C++ specific for a discussion of C++ typecast mechanisms.

1.1.1.2.4.8 C++ Reference Declarations

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

1.1.1.2.5 Arrays

The declaration

type declarator [<constant-expression>]

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

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

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

```
/* DYNAMIC MEMORY ALLOCATION FOR A MULTIDIMENSIONAL OBJECT. */
#include <stdio.h>
#include <stdlib.h>
typedef long double TYPE;
typedef TYPE *OBJECT;
unsigned int rows = 3, columns = 5;
void de_allocate(OBJECT);
int main(void) {
  OBJECT matrix;
  unsigned int i, j;
  /* STEP 1: SET UP THE ROWS. */
  matrix = (OBJECT) calloc( rows, sizeof(TYPE *));
  /* STEP 2: SET UP THE COLUMNS. */
  for (i = 0; i < rows; ++i)
    matrix[i] = (TYPE *) calloc( columns, sizeof(TYPE));
    for (i = 0; i < rows; i++)
      for (j = 0; j < columns; j++)
        matrix[i][j] = i + j;    /* INITIALIZE */
  for (i = 0; i < rows; ++i) {
    printf("\n\n");
    for (j = 0; j < columns; ++j)
      printf("%5.2Lf", matrix[i][j]);
  de_allocate(matrix);
  return 0;
  }
void de_allocate(OBJECT x) {
  int i;
```

```
  for (i = 0; i < rows; i++)    /* STEP 1: DELETE THE COLUMNS */
    free(x[i]);
  free(x);    /* STEP 2: DELETE THE ROWS. */
  }
```

This code produces the following output:

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

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

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

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

See Also

Arrays

1.1.1.2.6 Functions

This section contains Function topics.

1.1.1.2.6.1 Functions

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

Member functions are sometimes referred to as methods.

1.1.1.2.6.2 Declarations And Definitions

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

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

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

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

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

See Also

Functions (see page 55)

1.1.1.2.6.3 Declarations And Prototypes

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

<type> func()

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

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

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

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

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

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

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

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

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

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

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

```
func(void);
```

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

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

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

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

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

Here are some more examples of function declarators and prototypes:

```
int  f();/* In C, a function returning an int with no information about parameters.
This is the K&R "classic style." */
int f();/* In C++, a function taking no arguments */
int  f(void);/* A function returning an int that takes no parameters. */
int  p(int,long);/* A function returning an int thataccepts two parameters: the first, an int;
the second, a long. */
int  __pascal q(void);/* A pascal function returningan int that takes no parameters at all. */
int  printf(char *format,...;  /*A function returning an int andaccepting a pointer to a char
fixedparameter and any number of additionalparameters of unknown type. */
int  (*fp)(int)/* A pointer to a function returning an intand requiring an int parameter. */
```

1.1.1.2.6.4 Definitions

The general syntax for external function definitions is as follows:

External function definitions

```
file
    external-definition
     file  external-definition
external-definition:
    function-definition
    declaration
    asm-statement
function-definition:
    <declaration-specifiers> declarator <declaration-list>
    compound-statement
```

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

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

Note: You can mix elements from 1 and 2.

1.1.1.2.6.5 Formal Parameter Declarations

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

```
int func(void) {                    // no args
int func(T1 t1, T2 t2, T3 t3=1) {  // three simple parameters, one
                                    // with default argument
int func(T1* ptr1, T2& tref) {     // A pointer and a reference arg
int func(register int i) {         // Request register for arg
```

```
int func(char *str,...) {          /* One string arg with a variable number of other
                                   args, or with a fixed number of args with varying types */
```

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

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

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

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

1.1.1.2.6.6 Function Calls And Argument Conversions

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

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

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

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

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

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

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

1.1.1.2.7 Structures

This section contains Structure topics.

1.1.1.2.7.1 Structures

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

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

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

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

1.1.1.2.7.2 Untagged Structures And Typedefs

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

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

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

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

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

Untagged structure and union members are ignored during initialization.

1.1.1.2.7.3 Structure Member Declarations

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

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

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

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

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

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

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

Except in C++, a member cannot have the type "function returning...," but the type "pointer to function returning..." is allowed. In C++, a **struct** can have member functions.

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

1.1.1.2.7.4 Structures And Functions

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

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

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

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

1.1.1.2.7.5 Structure Member Access

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

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

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

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

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

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

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

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

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

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

the objects a and a1 are both of type struct A, but the objects a and b are of different structure types. Structures can be assigned

only if the source and destination have the same type:

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

1.1.1.2.7.6 Structure Name Spaces

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

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

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

1.1.1.2.7.7 Incomplete Declarations

Incomplete declarations are also known as forward declarations.

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

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

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

1.1.1.2.7.8 Bit Fields

Bit fields are specified numbers of bits that may or may not have an associated identifier. Bit fields offer a way of subdividing structures (structs, unions, classes) into named parts of user-defined sizes.

Declaring bit fields

You specify the bit-field width and optional identifier as follows:

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

In C++, type-specifier is **bool**, **char**, **unsigned char**, **short**, **unsigned short**, **long**, **unsigned long**, **int**, **unsigned int, __int64** or **unsigned __int64**. In strict ANSI C, type-specifier is **int** or **unsigned int**.

The expression width must be present and must evaluate to a constant integer. In C++, the width of a bit field may be declared of any size. In strict ANSI C, the width of a bit field may be declared only up to the size of the declared type. A zero length bit field skips to the next allocation unit.

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

Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (. and ->) used for non-bit-field members.

Limitations of using bit fields

When using bit fields, be aware of the following issues:

- The code will be non-portable since the organization of bits-within-bytes and bytes-within-words is machine dependent.
- You cannot take the address of a bit field; so the expression &mystruct.x is illegal if x is a bit field identifier, because there is no guarantee that mystruct.x lies at a byte address.
- Bit fields are used to pack more variables into a smaller data space, but causes the compiler to generate additional code to manipulate these variables. This costs in terms of code size and execution time.

Because of these disadvantages, using bit fields is generally discouraged, except for certain low-level programming. A recommended alternative to having one-bit variables, or flags, is to use defines. For example:

#define Nothing 0x00

#define bitOne 0x01

#define bitTwo 0x02

#define bitThree 0x04

#define bitFour 0x08

#define bitFive 0x10

#define bitSix 0x20

#define bitSeven 0x40

#define bitEight 0x80

can be used to write code like:

```
if (flags & bitOne) {...}    // is bit One turned on
flags |= bitTwo;               // turn bit Two on
flags &= ~bitThree;          // turn bit Three off
```

Similar schemes can be made for bit fields of any size.

Padding of bit fields

In C++, if the width size is larger than the type of the bit field, the compiler will insert padding equal to the requested width size minus the size of the type of the bit field. So, declaration:

```
struct mystruct
{
  int i : 40;
  int j : 8;
};
```

will create a 32 bit storage for 'i', an 8 bit padding, and 8 bit storage for 'j'. To optimize access, the compiler will consider 'i' to be a regular int variable, not a bit field.

Layout and alignment

Bit fields are broken up into groups of consecutive bit fields of the same type, without regard to signedness. Each group of bit fields is aligned to the current alignment of the type of the members of the group. This alignment is determined by the type AND by the setting of the overall alignment (set by the byte alignment option –aN). Within each group, the compiler will pack the bit fields inside of areas as large as the type size of the bit fields. However, no bit field is allowed to straddle the boundary between 2 of those areas. The size of the total structure will be aligned, as determined by the current alignment.

Example of bit field layout, padding, and alignment

In the following C++ declaration, my_struct contains 6 bit fields of 3 different types, **int**, **long**, and **char**:

```
struct my_struct
{
   int one : 8;
   unsigned int two : 16;
   unsigned long three : 8;
   long four : 16;
   long five : 16;
   char six : 4;
};
```

Bit fields 'one' and 'two' will be packed into one 32-bit area.

Next, the compiler inserts padding, if necessary, based on the current alignment, and the type of three, because the type changes between the declarations of variables two and three. For example, if the current alignment is byte alignment (-a1), no padding is needed; whereas, if the alignment is 4 bytes (-a4), then 8-bit padding is inserted.

Next, variables three, four, and five are all of type **long**. Variables three and four are packed into one 32 bit area, but five can not be packed into that same area, since that would create an area of 40 bits, which is more than the 32 bit allowed for the **long** type. To start a new area for five, the compiler would insert no padding if the current alignment is byte alignment, or would insert 8 bits of padding if the current alignment is dword (4 byte) alignment.

With variable six, the type changes again. Since **char** is always byte aligned, no padding is needed. To force alignment for the whole struct, the compiler will finish up the last area with 4 bits of padding if byte alignment is used, or 12 bits of padding if dword alignment is used.

The total size of my_struct is 9 bytes with byte alignment, or 12 bytes with dword alignment.

To get the best results when using bit fields you should:

- Sort bit fields by type
- Make sure they are packed inside the areas by ordering them such that no bit field will straddle an area boundary
- Make sure the struct is filled as much as possible.

Another recommendation is to force byte alignment for this struct, by emitting "#pragma option -a1". If you want to know how big your struct is, follow it by "#pragma sizeof(mystruct)", which gives you the size.

Using one bit signed fields

For a signed type of one bit, the possible values are 0 or –1. For an unsigned type of one bit, the possible values are 0 or 1. Note that if you assign “1” to a signed bit field, the value will be evaluated as –1 (negative one).

When storing the values **true** and **false** into a one bit sized bit field of a signed type, you can not test for equality to **true** because signed one bit sized bit fields can only hold the values '0' and '-1', which are not compatible with **true** and **false**. You can, however, test for non-zero.

For unsigned varieties of all types, and of course for the **bool** type, testing for equality to **true** will work as expected.

Thus:

```
struct mystruct
{
   int flag : 1;
```

```
} M;
int testing()
{
   M.flag = true;
   if(M.flag == true)
     printf("success");}
```

will NOT work, but:

```
struct mystruct
{
   int flag : 1;
} M;
int testing()
{
   M.flag = true;
   if(M.flag)
     printf("success");
}
```

works just fine.

Notes on compatibility

Between versions of the compiler, changes can be made to default alignment, or for purposes of compatibility with other compilers. Consequently, this could change the alignment of bit fields. Therefore, there is no guarantee that the alignment of bit fields will be consistent between versions of the compiler. To check the backward compatibility of bit fields in your code you can add an assert statement that checks for the structure size you are expecting.

According to the C and C++ language specifications, the alignment and storage of bit fields is implementation defined. Therefore, compilers can align and store bit fields differently. If you want complete control over the layout of bit fields, it is advisable to write your own bit field accessing routines and create your own bit fields.

1

1.1.1.2.8 Unions

This section contains Union topics.

1.1.1.2.8.1 Unions

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

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

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

Note: Unions correspond to the variant record types of Delphi.

sizeof(**union** myunion) and **sizeof**(mu) both return 8, but 4 bytes are unused (padded) when mu holds an **int** object, and 7 bytes are unused when mu holds a **char**. You access union members with the structure member selectors (. and ->), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); //OK: displays mu.d = 4.016printf("mu.i = %d\n",mu.i); //peculiar
resultmu.ch = 'A';
```

```
printf("mu.ch = %c\n",mu.ch); //OK: displays mu.ch = Aprintf("mu.d = %f\n",mu.d); //peculiar
resultmuptr->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.

1.1.1.2.8.2 Anonymous Unions

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

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

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

Anonymous unions can be global, nested, or unnested. ANSI-C never allows anonymous unions. ANSI-C++ allows all three types of anonymous unions.

C++ Anonymous unions

An anonymous union cannot have member functions or private or protected members. At file level an anonymous union must be declared static. Local anonymous unions must be either automatic or static; in other words, an anonymous union cannot have external linkage. Unnested anonymous unions are only allowed in C++.

Nested anonymous unions

The outer structure, class, or union of a nested anonymous union must have a tag. CodeGear C and C++ allow nested anonymous unions by default. In C only, a nested anonymous union can, optionally, have a tag.

Example

```
struct outer {
        int x;
};
int main(void)
{
        struct outer o;
}
```

1.1.1.2.8.3 Union Declarations

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

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

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

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

1.1.1.2.9 Enumerations

This section contains Enumeration topics.

1.1.1.2.9.1 Enumerations

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

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

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

The -b compiler switch controls the “Treat Enums As Ints” option. When this switch is used, the compiler allocates a whole word (a four-byte **int**) for enumeration types (variables of type **enum**). The default is ON (meaning **enums** are always **int**s) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, then that is the type of each enumerator.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
*daysptr = anyday;   // OK
mon = tues;          // ILLEGAL: mon is a constant
```

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

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

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

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

1.1.1.2.9.2 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).

1.1.1.2.10 Expressions

This section contains Expression topics.

1.1.1.2.10.1 Expressions

An expression is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in

CodeGear C++ expressions, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

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

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

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

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

CodeGear C++ expressions

```
primary-expression:
    literal
    this (C++ specific)
    :: identifier (C++ specific)
    :: operator-function-name (C++ specific)
    :: qualified-name (C++ specific)
    (expression)
    name

teral:
    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-expressionequality-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
```

1.1.1.2.10.2 Precedence Of Operators

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

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

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

Operators	Associativity
() [] -> :: .	left to right
! ~ + - ++ -- & * 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

1.1.1.2.10.3 Expressions And C++

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

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

1.1.1.2.10.4 Evaluation Order

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

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

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

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

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

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

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

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

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

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

1.1.1.2.10.5 Errors And Overflows

Associativity and precedence of CodeGear C++ operators. summarizes the precedence and associativity of the operators. During the evaluation of an expression, the compiler can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2n 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.

1.1.1.2.11 Operators Summary

This section contains Operator Summary topics.

1.1.1.2.11.1 Operators Summary

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

Arithmetic (see page 78)

Assignment (see page 84)

Bitwise (see page 81)

C++ specific (see page 85)

Comma (see page 84)

Conditional (see page 83)

Equality (see page 83)

Logical (see page 83)

Postfix Expression Operators

Primary Expression Operators (see page 73)

Preprocessor

Reference/Indirect operators (see page 76)

Relational (see page 82)

sizeof (see page 79)

typeid (see page 176)

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.

See Also

Precedence Of Operators (see page 70)

1.1.1.2.12 Primary Expression Operators

This section contains Primary Expression Operator topics.

1.1.1.2.12.1 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:

1. literal **this** (C++ specific)
2. **::** identifier (C++ specific)
3. **::** operator-function-name (C++ specific)
4. **::** qualified-name (C++ specific)
5. (expression) name

literal:

1. integer-constantcharacter-constant
2. floating-constant
3. string-literal

name:

1. identifieroperator-function-name (C++ specific)
2. conversion-function-name (C++ specific)
3. **~** class-name (C++ specific)

4. qualified-name (C++ specific)

qualified-name: (C++ specific)

qualified-class-name **::** name

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

The discussion of the scope resolution operator :: begins on page 104. The scope resolution operator allows reference to a type, object, function, or enumerator even though its identifier is hidden.

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

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

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

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

1.1.1.2.13 Postfix Expression Operators

This section contains Postfix Expression Operator topics.

1.1.1.2.13.1 Array Subscript Operator

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

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

is defined as

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

where either:

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

1.1.1.2.13.2 Function Call Operator

Syntax

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

Remarks

Parentheses ()

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

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

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

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

1.1.1.2.13.3 . (direct Member Selector)

Syntax

```
postfix-expression . identifier
```

postfix-expression must be of type union or structure.

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

Remarks

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

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

```
s.m
```

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

1.1.1.2.13.4 -> (indirect Member Selector)

Syntax

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

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

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

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

Remarks

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

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

```
sptr->m
```

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

The expression

```
s->sptr
```

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

1.1.1.2.13.5 Increment/decrement Operators

Increment operator (++)

Syntax

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

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

Postincrement operator

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

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

Preincrement operator

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

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

Pointer types follow the rules for pointer arithmetic.

Decrement operator (--)

Syntax

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

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

1.1.1.2.14 Unary Operators

This section contains Unary Operator topics.

1.1.1.2.14.1 Unary Operators

Syntax

<unary-**operator**> <unary expression>

OR

<unary-**operator**> <type><unary expression>

Remarks

Unary operators group right-to-left.

The C++ language provides the following unary operators:

- ! Logical negation
- * Indirection
- ++ Increment
- ~ Bitwise complement
- -- Decrement
- - Unary minus
- + Unary plus

1.1.1.2.14.2 Reference/deference Operators

Syntax

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

Remarks

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

Referencing operator (&)

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

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

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

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

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

Consider the following example:

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

T *ptr = &t1 is treated as

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

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

Indirection operator (*)

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

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

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

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

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

Note: & can also be the bitwise AND operator.

Note:
- can also be the multiplication operator.

1.1.1.2.14.3 Plus And Minus Operators

Unary

In these unary + - expressions

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

the cast-expression operand must be of arithmetic type.

Results

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

Binary

Syntax

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

Legal operand types for op1 + op2:

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

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

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

Legal operand types for op1 - op2:

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

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

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

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

1.1.1.2.14.4 Arithmetic Operators

Syntax

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

Remarks

Use the arithmetic operators to perform mathematical computations.

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

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

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

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

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

1.1.1.2.14.5 sizeof

Category

Operators

Description

The **sizeof** operator has two distinct uses:

sizeof unary-expression

sizeof (type-name)

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals sizeof array/ sizeof array[0] .

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

You cannot use **sizeof** with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is size_t.

You can use **sizeof** in preprocessor directives; this is specific to CodeGear C++.

In C++, sizeof(classtype), where classtype is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Example

```
/*  USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>
struct st {
char *name;
int age;
double height;
};
struct st St_Array[]= {  /* AN ARRAY OF structs */
{ "Jr.",     4,  34.20 },  /* St_Array[0] */
{ "Suzie",  23,  69.75 },  /* St_Array[1] */
};
int main()
{
long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };
printf("\nNumber of elements in LD_Array = %d",
sizeof(LD_Array) / sizeof(LD_Array[0]));
/****  THE NUMBER OF ELEMENTS IN THE St_Array. ****/
printf("\nSt_Array has %d elements",
sizeof(St_Array)/sizeof(St_Array[0]));
/****  THE NUMBER OF BYTES IN EACH St_Array ELEMENT.  ****/
```

```
printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));
/****  THE TOTAL NUMBER OF BYTES IN St_Array.  ****/
printf("\nSt_Array=%d", sizeof(St_Array));
return 0;
}
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 16
St_Array= 32
```

1.1.1.2.15 Binary Operators

This section contains Binary Operator topics.

1.1.1.2.15.1 Binary Operators

These are the binary operators in CodeGear C++:

- Arithmetic + Binary plus (add)
- - Binary minus (subtract)
- * Multiply
- / Divide
- % Remainder (modulus)
- Bitwise << Shift left
- >> Shift right
- & Bitwise AND
- ^ Bitwise XOR (exclusive OR)
- | Bitwise inclusive OR
- Logical && Logical AND
- || Logical OR
- Assignment = Assignment
- *= Assign product
- /= Assign quotient
- %= Assign remainder (modulus)
- += Assign sum
- -= Assign difference
- <<= Assign left shift
- >>= Assign right shift
- &= Assign bitwise AND
- ^= Assign bitwise XOR
- |= Assign bitwise OR
- Relational < Less than
- > Greater than
- <= Less than or equal to

- >= Greater than or equal to
- == Equal to
- != Not equal to
- Component selection . Direct component selector
- -> Indirect component selector
- Class-member :: Scope access/resolution
- .* Dereference pointer to class member
- ->* Dereference pointer to class member
- Conditional ? : Actually a ternary operator for example,
- a ? x : y "if a then x else y"
- Comma , Evaluate

1.1.1.2.15.2 Multiplicative Operators

Syntax

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

Remarks

There are three multiplicative operators:

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

The usual arithmetic conversions are made on the operands.

(op1 * op2) Product of the two operands

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

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

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

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

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

Note: Rounding is always toward zero.

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

1.1.1.2.15.3 Bitwise Operators

Syntax

```
AND-expression &  equality-expression
exclusive-OR-expr ^ AND-expression
inclusive-OR-expr exclusive-OR-expression
```

```
~cast-expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

Remarks

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

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

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

A	B	A & B	A ^ B	A | B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

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

Note: >> is often overloaded to be the input operator in I/O expressions. << is often overloaded to be the output operator in I/O expressions.

1.1.1.2.15.4 Relational Operators

Syntax

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

Remarks

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

> greater than

< less than

>= greater than or equal

<= less than or equal

In the expression

E1 <**operator**> E2

the operands must follow one of these conditions:

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

1.1.1.2.15.5 Equality Operators

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

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

The syntax is

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

1.1.1.2.15.6 Logical Operators

Syntax

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

Remarks

Operands in a logical expression must be of scalar type.

&& logical AND; returns **true** only if both expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **false**, the second expression is not evaluated.

|| logical OR; returns **true** if either of the expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **true**, the second expression is not evaluated.

! logical negation; returns **true** if the entire expression evaluates to be nonzero, otherwise returns **false**. The expression !E is equivalent to (0 == E).

1.1.1.2.15.7 Conditional Operators

Syntax

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

Remarks

The conditional operator ?: is a ternary operator.

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

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

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

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

1.1.1.2.15.8 Assignment Operators

Syntax

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

Remarks

The assignment operators are:

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

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

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

The expression

```
E1 op= E2
```

has the same effect as

```
E1 = E1 op E2
```

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

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

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

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

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

Note: There are certain conditions where assignment operators are not supported when used with properties.

1.1.1.2.15.9 Comma Operator

Syntax

```
expression , assignment-expression
```

Remarks

The comma separates elements in a function argument list.

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

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

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

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

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

```
func(i, (j = 1, j + 4), k);
```

calls func with three arguments (i, 5, k), not four.

1.1.1.2.15.10 C++ Specific Operators

The operators specific to C++ are:

Operator	Meaning
::	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.

1.1.1.2.16 Statements

This section contains Statement topics.

1.1.1.2.16.1 Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. CodeGear C++ statements shows the syntax for statements.

CodeGear C++ statements

statement	labeled-statement compound-statement expression-statement selection-statement iteration-statement jump-statement asm-statement declaration (C++ specific)
labeled-statement:	identifier : statement **case** constant-expression : statement **default** : statement
compound-statement:	{ <declaration-list> <statement-list> }
declaration-list:	declaration declaration-list declaration
statement-list:	statement statement-list statement
expression-statement:	<expression> ;
asm-statement:	**asm** tokens newline **asm** tokens; **asm** { tokens; <tokens;>= <tokens;>}
selection-statement:	**if** (expression) statement **if** (expression) statement else statement **switch** (expression) statement
iteration-statement:	**while** (expression) statement **do** statement while (expression) ; **for** (for-init-statement <expression> ; <expression>) statement
for-init-statement:	expression-statement declaration (C++ specific)
jump-statement:	**goto** identifier ; **continue** ; **break** ; **return** <expression> ;

1.1.1.2.16.2 Blocks

A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces (**{ }**). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.

1.1.1.2.16.3 Labeled Statements

A statement can be labeled in two ways:

- label-identifier **:** statement

* The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.

- **case** constant-expression **:** statement

* **default :** statement

Case and default labeled statements are used only in conjunction with switch statements.

1.1.1.2.16.4 Expression Statements

Any expression followed by a semicolon forms an expression statement:

```
<expression>;
```

The compiler executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls

The null statement is a special case, consisting of a single semicolon (**;**). The null statement does nothing, and is therefore useful in situations where C++ syntax expects a statement but your program does not need one.

1.1.1.2.16.5 Selection Statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

1.1.1.2.16.6 Iteration Statements

Iteration statements let you loop a set of statements. There are three forms of iteration in C++: **while**, **do while**, and **for** loops.

1.1.1.2.16.7 Jump Statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**

1.1.1.3 C++ Specifics

This section contains C++ Specific topics.

1.1.1.3.1 C++ Specifics

C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function func declared twice in C with different argument types causes a duplicated name error. Under C++, however, func will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. This topic discusses the aspects of C++ that can be used independently of classes, then describes the specifics of classes and class mechanisms.

See C++ Exception Handling and C-Based Structured Exceptions for details on compiling C and C++ programs with exception handling.

See Also

Referencing (see page 95)

C++ Classes (see page 100)

Introduction To Constructors And Destructors (see page 118)

Polymorphic Classes (see page 135)

C++ Scope (see page 140)

1.1.1.3.2 C++ namespaces

This section contains C++ name space topics.

1.1.1.3.2.1 Defining A namespace

The grammar for defining a namespace is

```
original-namespace-name:
    identifier
 namespace-definition:
    original-namespace-definition
    extension-namespace-definition
    unnamed-namespace-definition
```

Grammatically, there are three ways to define a namespace with the **namespace** keyword:

```
original-namespace-definition:
    namespace identifier { namespace-body }
extension-namespace-definition:
    namespace original-namespace-name { namespace-body }
unnamed-namespace-definition:
    namespace { namespace-body }
```

The body is an optional sequence of declarations. The grammar is

```
namespace-body:
    declaration-seq opt
```

See Also

Declaring A namespace (see page 88)

Extending A namespace (see page 89)

1.1.1.3.2.2 Declaring A namespace

An original namespace declaration should use an identifier that has not been previously used as a global identifier.

```
namespace alpha {  /* ALPHA is the identifier of this namespace. */
  /* your program declarations */
  long double LD;
  float f(float y) { return y; }
  }
```

A namespace identifier must be known in all translation units where you intend to access its elements.

See Also

namespace Alias (see page 89)

1.1.1.3.2.3 namespace Alias

You can use an alternate name to refer to a namespace identifier. An alias is useful when you need to refer to a long, unwieldy namespace identifier.

```
namespace BORLAND_SOFTWARE_CORPORATION {
   /* namespace-body */
   namespace NESTED_BORLAND_SOFTWARE_CORPORATION {
      /* namespace-body */
      }
   }
// Alias namespace
namespace BI = BORLAND_SOFTWARE_CORPORATION;
// Use access qualifier to alias a nested namespace
namespace NBI = BORLAND_SOFTWARE_CORPORATION::NESTED_BORLAND_SOFTWARE_CORPORATION;
```

1.1.1.3.2.4 Extending A namespace

Namespaces are discontinuous and open for additional development. If you redeclare a namespace, the effect is that you extend the original namespace by adding new declarations. Any extensions that are made to a namespace after a using-declaration, will not be known at the point at which the using-declaration occurs. Therefore, all overloaded versions of some function should be included in the namespace before you declare the function to be in use.

1.1.1.3.2.5 Anonymous namespaces

The C++ grammar allows you to define anonymous namespaces. To do this, you use the keyword **namespace** with no identifier before the enclosing brace.

```
namespace {          // Anonymous namespace
   // Declarations
   }
```

All anonymous, unnamed namespaces in global scope (that is, unnamed namespaces that are not nested) of the same translation unit share the same namespace. This way you can make static declarations without using the **static** keyword.

Each identifier that is enclosed within an unnamed namespace is unique within the translation unit in which the unnamed namespace is defined.

1.1.1.3.2.6 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 (see page 90)
- Using directive (see page 90)
- Using declaration (see page 177)

Accessing namespaces in classes

You cannot use a **using** directive inside a class. However, the **using** declarative is allowed and can be quite useful.

See Also

Using Directive (see page 90)

Using (declaration) (see page 177)

1.1.1.3.2.7 using Directive

If you want to use several (or all of) the members of a namespace, C++ provides an easy way to get access to the complete namespace. The using-directive causes all identifiers in a namespace to be in scope at the point that the using-directive statement is made. The grammar for the using-directive is as follows.

using-directive:

using namespace :: opt nested-name-specifier opt namespace-name;

The using-directive is transitive. When you apply the using directive to a namespace that contains using directives within itself, you get access to those namespaces as well. For example, if you apply the using directive in your program, you also get namespaces `qux`, `foo`, and `bar`.

```
namespace qux {
   using namespace foo;  // This has been defined previously
   using namespace bar;  // This also has been defined previously
   }
```

The using-directive does not add any identifiers to your local scope. Therefore, an identifier defined in more than one namespace won't be a problem until you actually attempt to use it. Local scope declarations take precedence by hiding all other similar declarations.

Warning: Do not use the **using** directive in header files. You might accidentally break namespaces in client code.

See Also

Accessing Elements Of A namespace (see page 89)

Using (declaration) (see page 177)

1.1.1.3.2.8 Explicit Access Qualification

You can explicitly qualify each member of a namespace. To do so, you use the namespace identifier together with the **::** scope resolution operator followed by the member name. For example, to access a specific member of namespace ALPHA, you write:

```
ALPHA::LD;  // Access a variable
ALPHA::f;   // Access a function
```

Explicit access qualification can always be used to resolve ambiguity. No matter which namespace (except anonymous namespace) is being used in your subsystem, you can apply the scope resolution operator **::** to access identifiers in any namespace (including a namespace already being used in the local scope) or the global namespace. Therefore, any identifier in the application can be accessed with sufficient qualification.

New-style typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

1.1.1.3.3 New-style Typecasting Overview

This section contains New-style Typecasting Overview topics.

1.1.1.3.3.1 New-style Typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier

cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

See Also

Const_cast (typecast Operator) (see page 91)

Dynamic_cast (typecast Operator) (see page 91)

Reinterpret_cast (typecast Operator) (see page 92)

1.1.1.3.3.2 const_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
const_cast< T > (arg)
```

Description

Use the **const_cast** operator to add or remove the **const** or **volatile** modifier from a type.

In the statement, const_cast< T > (arg), T and arg must be of the same type except for **const** and **volatile** modifiers. The cast is resolved at compile time. The result is of type T. Any number of const or volatile modifiers can be added or removed with a single **const_cast** expression.

A pointer to **const** can be converted to a pointer to non-**const** that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A **const** object or a reference to **const** cast results in a non-**const** object or reference that is otherwise an identical type.

The **const_cast** operator performs similar typecasts on the **volatile** modifier. A pointer to volatile object can be cast to a pointer to non-**volatile** object without otherwise changing the type of the object. The result is a pointer to the original object. A **volatile**-type object or a reference to **volatile**-type can be converted into an identical non-**volatile** type.

See Also

Dynamic_cast (typecast Operator) (see page 91)

Reinterpret_cast (typecast Operator) (see page 92)

1.1.1.3.3.3 dynamic_cast (typecast Operator)

Category

C++-Specific Keywords

Description

In the expression, dynamic_cast< T > (ptr), T must be a pointer or a reference to a defined class type or **void***. The argument ptr must be an expression that resolves to a pointer or reference.

If T is void* then ptr must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if T is a pointer and ptr is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If T is a reference and ptr is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at runtime.

If successful, dynamic_cast< T > (ptr) converts ptr to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the Bad_cast exception is thrown.

Note: Runtime type identification (RTTI) is required for **dynamic_cast**.

See Also

Const_cast (typecast Operator) (see page 91)

Reinterpret_cast (typecast Operator) (see page 92)

__rtti (see page 94)

Static_cast (typecast Operator) (see page 92)

1.1.1.3.3.4 reinterpret_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
reinterpret_cast< T > (arg)
```

Description

In the statement, reinterpret_cast< T > (arg), T must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral arg can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

See Also

Const_cast (typecast Operator) (see page 91)

Dynamic_cast (typecast Operator) (see page 91)

1.1.1.3.3.5 static_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
static_cast< T > (arg)
```

Description

In the statement, static_cast< T > (arg), T must be a pointer, reference, arithmetic type, or enum type. Both T and arg must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using **static_cast** achieves exactly the same thing.

Integral types can be converted to **enum** types. A request to convert arg to a value that is not an element of **enum** is undefined.

The null pointer is converted to the null pointer value of the destination type, T.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class X to a pointer to some class Y if X is a base class for Y. A static conversion can be made only under the following conditions:

- if an unambiguous conversion exists from Y to X
- if X is not a virtual base class

An object can be explicitly converted to a reference type X& if a pointer to that object can be explicitly converted to an X*. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.

An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.

A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When T is a reference the result of static_cast< T > (arg) is an lvalue. The result of a pointer or reference cast refers to the original expression.

See Also

Const_cast (typecast Operator) (see page 91)

Dynamic_cast (typecast Operator) (see page 91)

Reinterpret_cast (typecast Operator) (see page 92)

1.1.1.3.4 Run-time Type Identification (RTTI)

This section contains Run-time Type Identification (RTTI) topics.

1.1.1.3.4.1 Runtime Type Identification (RTTI) Overview

Runtime type identification (RTTI) lets you write portable code that can determine the actual type of a data object at runtime even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. Use the dynamic_cast operator to make runtime casts.

The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with typeid operator, which determines the actual type of its argument and returns a reference to an object of type **const** type_info, which describes that type.

You can also use a type name as the argument to **typeid**, and **typeid** will return a reference to a **const** type_info object for that type. The class type_info provides an **operator==** and an **operator!=** that you can use to determine whether two objects are of the same type. Class type_info also provides a member function name that returns a pointer to a character string that holds the name of the type.

See Also

__rtti (see page 94)

The Typeid Operator (see page 176)

1.1.1.3.4.2 The Typeid Operator

This section contains Typeid Operator topics.

1.1.1.3.4.2.1 __rtti, -RT Option

Category (__rtti keyword)

Modifiers, C++ Keyword Extensions, C++-Specific Keywords

Description

Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the **-RT-** option to disable it or **-RT** to enable it.

If RTTI is disabled, or if the argument to **typeid** is a pointer or a reference to a non-polymorphic class, **typeid** returns a reference to a **const** type_info object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic runtime type identification (where appropriate) by using the **__rtti** keyword in the class definition.

When runtime type identification is disabled, if any base class is declared **__rtti**, then all polymorphic base classes must also be declared **__rtti**.

```
struct __rtti S1 { virtual s1func(); };  /* Polymorphic */
struct __rtti S2 { virtual s2func(); };  /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism, type information might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations without RTTI, you should declare X with the **__rtti** modifier. Otherwise, switching the order of the base classes for the class X results in the compile-time error "Can't inherit non-RTTI class from RTTI base 'S1'."

```
struct __rtti S1 { virtual func(); };   /* Polymorphic class */
struct S2 { };                           /*  Non-polymorphic class */
struct __rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with **__rtti**. This makes it safe to mix the order and type of classes.

In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared **__rtti**.

```
struct __rtti S1 {  };   // Non-polymorphic class
struct S2 { };
struct X : S2, S1 { };   // The order is not essential
```

Neither the **__rtti** keyword, nor enabling RTTI will make a static class into a polymorphic class.

See Also

Runtime Type Identification (RTTI) Overview (see page 93)

Runtime Type Identification And Destructors (see page 95)

1.1.1.3.4.2.2 Runtime Type Identification And Destructors

When destructor cleanup is enabled, a pointer to a class with a virtual destructor can't be deleted if that class is not compiled with runtime type identification enabled. The runtime type identification and destructor cleanup options are on by default. They can be disabled from the C++ page of the Project Options dialog box, or by using the **-xd-** and **-RT-** command-line options.

Example

```
class Alpha {
public:
   virtual ~Alpha( ) { }
};
void func( Alpha *Aptr ) {
   delete Aptr;         // Error.  Alpha is not a polymorphic class type
}
```

1.1.1.3.5 Referencing

This section contains Typeid Operator topics.

1.1.1.3.5.1 Referencing

In the C programming language, you can pass arguments only by value. In C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects. See the following topics for a discussion of referencing.

Simple references (see page 95)

Reference arguments (see page 95)

Note: C++ specific pointer referencing and dereferencing is discussed in C++ specific operators.

See Also

Simple References (see page 95)

Reference Arguments (see page 95)

1.1.1.3.5.2 Simple References

The reference declarator can be used to declare references outside functions:

```
int  i  = 0;
int &ir = i;   // ir is an alias for i
ir = 2;        // same effect as i = 2
```

This creates the lvalue ir as an alias for i, provided the initializer is the same type as the reference. Any operations on ir have precisely the same effect as operations on i. For example, ir = 2 assigns 2 to i, and &ir returns the address of i.

See Also

Referencing (see page 95)

Reference Arguments (see page 95)

1.1.1.3.5.3 Reference Arguments

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir);   // ir is type "reference to int"
```

```
    .
    .
    .
int sum = 3;
func1(sum);             // sum passed by value
func2(sum);             // sum passed by reference
```

The sum argument passed by reference can be changed directly by func2. On the other hand, func1 gets a copy of the sum argument (passed by value), so sum itself cannot be altered by func1.

When an actual argument x is passed by value, the matching formal argument in the function receives a copy of x. Any changes to this copy within the function body are not reflected in the value of x outside the scope of the function. Of course, the function can return a value that could be used later to change x, but the function cannot directly alter a parameter passed by value.

In C, changing the value of a function parameter outside the scope of the function requires that you pass the address of the parameter. The address is passed by value, thus changing the contents of the address effects the value of the parameter outside the scope of the function.

Even if the function does not need to change the value of a parameter, it is still useful to pass the address (or a reference) to a function. This is especially true if the parameter is a large data structure or object. Passing an object directly to a function necessitates copying the entire object.

Compare the three implementations of the function treble:

Implementation 1

```
int treble_1(int n)
{
   return 3 * n;
}
  .
  .
  .
int x, i = 4;
x = treble_1(i);  // x now = 12, i = 4
  .
  .
  .
```

Implementation 2

```
void treble_2(int* np)
{
   *np = (*np) * 3;
}
  .
  .
  .
treble_2(&i);    // i now = 2
```

Implementation 3

```
void treble_3(int& n)   // n is a reference type
{
   n = n * 3;
}
  .
  .
  .
treble_3(i);             // i now = 36
```

The formal argument declaration **type**& t establishes t as type “reference to type.” So, when treble_3 is called with the real argument i, i is used to initialize the formal reference argument n. n therefore acts as an alias for i, so n = n*3 also assigns 3 * i to i.

If the initializer is a constant or an object of a different type than the reference type, creates a temporary object for which the

reference acts as an alias:

```
int& ir = 6;   /* temporary int object created, aliased by ir, gets value 6 */
float f;
int& ir2 = f;  /* creates temporary int object aliased by ir2; f converted
                  before assignment */
ir2 = 2.0      // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

See Also

Referencing (see page 95)

Simple References (see page 95)

1.1.1.3.6 The Scope Resolution Operator

This section contains Scope Resolution Operator topics.

1.1.1.3.6.1 Scope Resolution Operator ::

The scope access (or resolution) operator **::** (two colons) lets you access a global (or file duration) member name even if it is hidden by a local redeclaration of that name. You can use a global identifier by prefixing it with the scope resolution operator. You can access a nested member name by specifying the class name and using the scope resolution operator. Therefore, Alpha::func( ) and Beta::func( ) are two different functions.

1.1.1.3.7 The new And delete Operators

This section contains new And delete Operator topics.

1.1.1.3.7.1 new

Category

Operators, C++-Specific Keywords

Syntax

```
void *operator new(std::size_t size) throw(std::bad_alloc);
void *operator new(std::size_t size, const std::nothrow_t &) throw();
void *operator new[](std::size_t size) throw(std::bad_alloc);
void *operator new[](std::size_t size, const std::nothrow_t &) throw();
void *operator new(std::size_t size, void *ptr) throw();    // Placement form
void *operator new[](std::size_t size, void *ptr) throw();  // Placement form
```

Description

The **new** operators offer dynamic storage allocation, similar but superior to the standard library function malloc. These allocation functions attempt to allocate size bytes of storage. If successful, **new** returns a pointer to the allocated memory. If the allocation fails, the **new** operator will call the new_handler function. The default behavior of new_handler is to throw an exception of type bad_alloc. If you do not want an exception to be thrown, use the nothrow version of **operator new**. The nothrow versions return a null pointer result on failure, instead of throwing an exception.

The default placement forms of **operator new** are reserved and cannot be redefined. You can, however, overload the placement form with a different signature (i.e. one having a different number, or different type of arguments). The default placement forms

accept a pointer of type void, and perform no action other than to return that pointer, unchanged. This can be useful when you want to allocate an object at a known address. Using the placement form of **new** can be tricky, as you must remember to explicitly call the destructor for your object, and then free the pre-allocated memory buffer. Do not call the **delete** operator on an object allocated with the placement **new** operator.

A request for non-array allocation uses the appropriate **operator new()** function. Any request for array allocation will call the appropriate **operator new[]()** function. Remember to use the array form of **operator delete[]()**, when deallocating an array created with **operator new[]()**.

Note: Arrays of classes require that a default constructor be defined in the class.

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

See Also

Delete (see page 98)

The Operator New With Arrays (see page 99)

Operator New (see page 99)

Operator New Placement Syntax (see page 99)

Handling Errors For The New Operator (see page 99)

Example Of Using Operator new With Nothrow

1.1.1.3.7.2 delete

Category

Operators, C++-Specific Keywords

Syntax

```
void operator delete(void *ptr) throw();
void operator delete(void *ptr, const std::nothrow_t&) throw();
void operator delete[](void *ptr) throw();
void operator delete[](void *ptr, const std::nothrow_t &) throw();
void operator delete(void *ptr, void *) throw();   // Placement form
void operator delete[](void *ptr, void *) throw(); // Placement form
```

Description

The **delete** operator deallocates a memory block allocated by a previous call to new. It is similar but superior to the standard library function free.

You should use the **delete** operator to remove all memory that has been allocated by the **new** operator. Failure to free memory can result in memory leaks.

The default placement forms of **operator delete** are reserved and cannot be redefined. The default placement **delete** operator performs no action (since no memory was allocated by the default placement **new** operator). If you overload the placement version of operator new, it is a good idea (though not strictly required) to provide the overload the placement delete operator with the corresponding signature.

See Also

Overloading The Operator delete (see page 100)

The Delete Operator With Arrays (see page 99)

1.1.1.3.7.3 Operator new Placement Syntax

The placement syntax for **operator new( )** can be used only if you have overloaded the allocation operator with the appropriate arguments. You can use the placement syntax when you want to use and reuse a memory space which you set up once at the beginning of your program.

When you use the overloaded **operator new( )** to specify where you want an allocation to be placed, you are responsible for deleting the allocation. Because you call your version of the allocation operator, you cannot depend on the global **::operator delete( )** to do the cleanup.

To release memory, you make an explicit call on the destructor. This method for cleaning up memory should be used only in special situations and with great care. If you make an explicit call of a destructor before an object that has been constructed on the stack goes out of scope, the destructor will be called again when the stackframe is cleaned up.

1.1.1.3.7.4 Handling Errors For The New Operator

By default, **new** throws the bad_alloc exception when a request for memory allocation cannot be satisfied.

You can define a function to be called if the **new** operator fails. To tell the **new** operator about the new-handler function, use set_new_handler and supply a pointer to the new-handler. If you want **new** to return null on failure, you must use set_new_handler(0) .

See Also

New (see page 97)

1.1.1.3.7.5 The Operator new With Arrays

When using the array form of operator new[](), the pointer returned points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];    // OK
mat_ptr = new int[n][10][12];    // OK
mat_ptr = new int[3][][12];      // illegal
mat_ptr = new int[][10][12];     // illegal
```

Although the first array dimension can be a variable, all following dimensions must be constants.

1.1.1.3.7.6 The delete Operator With Arrays

Arrays are deleted by operator delete[](). You must use the syntax **delete []** expr when deleting an array.

```
char * p;
void func()
{
   p = new char[10];    //  allocate 10 chars
   delete[] p;          // delete 10 chars
}
```

Early C++ compilers required the array size to be named in the delete expression. In order to handle legacy code, the compiler issues a warning and simply ignores any size that is specified. For example, if the preceding example reads delete[10] p and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

1.1.1.3.7.7 Operator new

By default, if there is no overloaded version of new, a request for dynamic memory allocation always uses the global version of

new, **::operator new()**. A request for array allocation calls **::operator new[]()**. With class objects of type name, a specific operator called name**::operator new()** or name**::operator new[]()** can be defined. When **new** is applied to class name objects it invokes the appropriate name**::operator new** if it is present; otherwise, the global **::operator new** is used.

Only the **operator new()** function will accept an optional initializer. The array allocator version, **operator new[](),** will not accept initializers. In the absence of explicit initializers, the object created by new contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the default constructor. The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

See Also

Overloading The Operator New (see page 100)

1.1.1.3.7.8 Overloading The Operator new

The global ::operator new() and **::operator new[]()** can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program.

Class-specific memory allocation operators can also be overloaded. The operator **new** can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator **new** must return a **void*** and must have a size_t as its first argument. To overload the **new** operators, use the following prototypes declared in the new.h header file.

- void * operator new(size_t Type_size); // For Non-array
- void * operator new[](size_t Type_size); // For arrays

The compiler provides Type_size to the **new** operator. Any data type may be substituted for Type_size except function names (although a pointer to function is permitted), class declarations, enumeration declarations, const, volatile.

1.1.1.3.7.9 Overloading The Operator delete

The global operators, ::operator delete(), and **::operator delete[]()** cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the each global delete function can exist in the program.

The user-defined operator delete must have a **void** return type and **void*** as its first argument; a second argument of type size_t is optional. A class T can define at most one version of each of T**::operator delete[]()** and T**::operator delete()**. To overload the **delete** operators, use the following prototypes.

- void operator delete(void *Type_ptr, [size_t Type_size]); // For Non-array
- void operator delete[](size_t Type_ptr, [size_t Type_size]); // For arrays

1.1.1.3.8 Classes

This section contains Class topics.

1.1.1.3.8.1 C++ Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that inherit the members of one or more base (or parent) classes.

In C++, structures and unions are considered classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

class-key <type-info> class-name

<: base-list> { <member-list> };

class-key is one of **class**, **struct**, or **union**.

The optional type-info indicates a request for runtime type information about the class. You can compile with the **–RT** compiler option, or you can use the __**rtti** keyword.

The optional base-list lists the base class or classes from which the class class-name will derive (or inherit) objects and methods. If any base classes are specified, the class class-name is called a derived class. The base-list has default and optional overriding access specifiers that can modify the access rights of the derived class to members of the base classes.

The optional member-list declares the class members (data and functions) of class-name with default and optional overriding access specifiers that can affect which functions can access which members.

See Also

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.2 VCL Class Declarations

Syntax

```
__declspec(<decl-modifier>)
```

Description

The decl-modifier argument can be **delphiclass** or **pascalimplementation**. These arguments should be used only with classes derived from VCL classes.

- You must use __declspec(delphiclass) for any forward declaration of classes that are directly or indirectly derived from TObject.

- Use the __declspec(pascalimplementation) modifier to indicate that a class has been implemented in the Delphi language. This modifier appears in a Delphi portability header file with a .hpp extension.

Note: Another argument, **delphireturn**, is used internally to mark C++ classes for VCL-compatible handling in function calls as parameters and return values.

The **delphiclass** argument is used to create classes that have the following VCL compatibility.

- VCL-compatible RTTI
- VCL-compatible constructor/destructor behavior
- VCL-compatible exception handling

A VCL-compatible class has the following restrictions.

- No virtual base classes or multiple inheritance is allowed.
- Must be dynamically allocated by using the global **new** operator.
- Copy and assignment constructors must be explicitly defined. The compiler does not automatically provide these constructors for VCL-derived classes.
- Must publicly inherit from another VCL class.

See Also

VCL Class Names

1.1.1.3.8.3 Class Names

class-name is any identifier unique within its scope. With structures, classes, and unions, class-name can be omitted. See Untagged structures and typedefs for discussion of untagged structures.

See Also

C++ Classes (see page 100)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

VCL Class Names

1.1.1.3.8.4 Class Types

The declaration creates a unique type, class type class-name. This lets you declare further class objects (or instances) of this type, and objects derived from this type (such as pointers to, references to, arrays of class-name, and so on):

```
class X { ... };
X x, &xr, *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)

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.5 Class Name Scope

The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the elaborated type specifier. This means that the class key, **class**, **struct**, or **union**, must be used with the class name. For example,

```
struct S { ... };
int S(struct S *Sptr);
void func(void) {
   S t;         // ILLEGAL declaration: no class key and function S in scope
```

```
    struct S s;  // OK: elaborated with class key
    S(&s);       // OK: this is a function call
}
```

C++ also allows a forward class declaration:

```
class X;  // no members, yet!
```

Forward declarations permit certain references to class name X (usually references to pointers to class objects) before the class has been fully defined. See Structure member declarations for more information. Of course, you must make a complete class declaration with members before you can define and use objects of that class.

See also the syntax for forward declarations of VCL classes.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.6 Class Objects

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class.

Redefined functions and operators are said to be overloaded. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called member functions for that class. C++ offers the overloading mechanism that allows the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.7 Class Member List

The optional member-list is a sequence including, but not exclusive to:

- Data declarations (of any type, including enumerations, bit fields and other classes)
- Nested type declarations
- Nested type definitions
- Template declarations
- Template definitions
- Function declarations
- Function definitions
- Constructors
- A destructor

Members of a class can optional have storage class specifiers and access modifiers. The objects thus defined are called class members. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.8 Member Functions

A function declared without the **friend** specifier is known as a member function of the class. Functions declared with the **friend** modifier are called friend functions.

Member functions are often referred to as methods in Delphi documentation.

The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.9 The Keyword This

This section contains Keyword this topics.

1.1.1.3.8.9.1 Static Members

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called static members and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If x is a static member of class X, it can be referenced as X::x (even if objects of class X haven’t been created yet). It is still possible to access x using the normal member access operators. For example, y.x and yptr->x, where y is an object

of class X and yptr is a pointer to an object of class X, although the expressions y and yptr are not evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
   int member_int;
public:
   static void func(int i, X* ptr);
};
void g(void)
{
   X obj;
   func(1, &obj);      // error unless there is a global func()
                       // defined elsewhere
   X::func(1, &obj);   // calls the static func() in X
                       // OK for static functions only
   obj.func(1, &obj);  // so does this (OK for static and
                       // nonstatic functions)
}
```

Because static member functions can be called with no particular object in mind, they don't have a **this** pointer, and therefore cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, func might be defined as follows:

```
void X::func(int i, X* ptr)
{
   member_int = i;       // which object does member_int
                         // refer to? Error
   ptr->member_int = i;  // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members, nested to any level, obey the usual class member access rules, except they can be initialized.

```
class X {
   static int x;
   static const int size =  5;
   class inner {
      static float f;
      void func(void);     // nested declaration
      };
public :
   char array[size];
};
int X::x = 1;
float X::inner::f = 3.14;  // initialization of nested static
void X::inner::func(void) {     /*  define the nested function */  }
```

The principal use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- Reduce the number of visible global names
- Make obvious which static objects logically belong to which class
- Permit access control to their names

See Also

C++ Classes (see page 100)

1.1.1.3.8.10 Inline Functions

This section contains Inline Function topics.

1.1.1.3.8.10.1 Inline Functions

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an inline function.

The compiler can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an inline expansion of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The **inline** specifier indicates to the compiler you would like an inline expansion.

Note: The compiler can ignore requests for inline expansion.

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the operator functions that implement overloaded operators. For example, the following class declaration of func:

```
int i;                          // global int
class X {
public:
   char* func(void) { return i; }  // inline by default
   char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined outside the class with an explicit **inline** specifier. The value i returned by func is the **char*** i of class X (see Member scope).

Inline functions and exceptions

An inline function with an exception-specification will never be expanded inline by the compiler. For example,

```
inline void f1() throw(int)
{
   // Warning: Functions with exception specifications are not expanded inline
}
```

The remaining restrictions apply only when destructor cleanup is enabled.

Note: Destructors are called by default. See Setting exception handling options for information about exception-handling switches.

An inline function that takes at least one parameter that is of type 'class with a destructor' will not be expanded inline. Note that this restriction does not apply to classes that are passed by reference. Example:

```
struct foo {
   foo();
   ~foo();
};
inline void f2(foo& x)
{
   // no warning, f2() can be expanded inline
}
inline void f3(foo x)
{
   // Warning: Functions taking class-by-value argument(s) are
   //          not expanded inline in function f3(foo)
}
```

An inline function that returns a class with a destructor by value will not be expanded inline whenever there are variables or temporaries that need to be destructed within the return expression:

```
struct foo {
   foo();
   ~foo();
};
inline foo f4()
{
   return foo();
   // no warning, f4() can be expanded inline
}
inline foo f5()
{
   foo X;
   return foo(); // Object X needs to be destructed
   // Warning: Functions containing some return statements are
   //          not expanded inline in function f5()
}
inline foo f6()
{
   return ( foo(), foo() );  // temporary in return value
   // Warning:Functions containing some return statements are
   //         not expanded inline in function f6()
}
```

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

1.1.1.3.8.11 Member Scope

This section contains Member Scope topics.

1.1.1.3.8.11.1 Member Scope

The expression X::func() in the example in Inline functions and exceptions uses the class name X with the scope access modifier to signify that func, although defined “outside” the class, is indeed a member function of X and exists within the scope of X. The influence of X:: extends into the body of the definition. This explains why the i returned by func refers to X::i, the **char*** i of X, rather than the global **int** i. Without the X:: modifier, the function func would represent an ordinary non-class function, returning the global **int** i.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class X can be referenced using the selection operators **.** and **->** (as with C structures). Member functions can also be called using the selection operators (see The keyword this). For example:

```
class X {
public:
   int i;
   char name[20];
   X *ptr1;
   X *ptr2;
   void Xfunc(char*data, X* left, X* right);   // define elsewhere
};
void f(void);
{
   X x1, x2, *xptr=&x1;
   x1.i = 0;
   x2.i = x1.i;
   xptr–>i = 1;
   x1.Xfunc("stan", &x2, xptr);
}
```

If m is a member or base member of class X, the expression X::m is called a qualified name; it has the same type as m, and it is an lvalue only if m is an lvalue. It is important to note that, even if the class name X is hidden by a non-type name, the qualified name X::m will access the correct class member, m.

Class members cannot be added to a class by another section of your program. The class X cannot contain objects of class X, but can contain pointers or references to objects of class X (note the similarity with C’s structure and union types).

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.11.2 Nested Types

Tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the **xxx::yyy** notation, except when in the scope of the appropriate class.

A class declared within another class is called a nested class. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).

Classes can be nested in this way to an arbitrary level, up to the limits of memory. Nested classes can be declared inside some class and defined later. For example,

```
struct outer
{
   typedef int t;  // 'outer::t' is a typedef name
   struct inner    // 'outer::inner' is a class
   {
      static int x;
   };
   static int x;
      int f();
   class deep;     // nested declaration
};
int outer::x;     // define static data member
int outer::f() {
   t x;           // 't' visible directly here
   return x;
   }
int outer::inner::x;    // define static data member
outer::t x;             //  have to use 'outer::t' here
class outer::deep { };  // define the nested class here
```

With CodeGear 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'
```

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.11.3 Member Access Control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

- **public**: The member can be used by any function.
- **private**: The member can be used only by member functions and friends of the class it's declared in.
- **protected**: Same as for **private**. Additionally, the member can be used by member functions and friends of classes derived from the declared class, but only in objects of the derived type. (Derived classes are explained in Base and derived class access.)

Note: Friend function declarations are not affected by access specifiers (see Friends of classes for more information).

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
   int i;    // X::i is private by default
   char ch;  // so is X::ch
public:
   int j;    // next two are public
   int k;
protected:
   int l;    // X::l is protected
};
struct Y {
   int i;    // Y::i is public by default
private:
   int j;    // Y::j is private
public:
   int k;    // Y::k is public
};
union Z {
   int i;    // public by default; no other choice
   double d;
};
```

Note: The access specifiers can be listed and grouped in any convenient sequence. You can save typing effort by declaring all the private members together, and so on.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.11.4 Base And Derived Class Access

When you declare a derived class D, you list the base classes B1, B2, ... in a comma-delimited base-list:

```
class-key D : base-list { <member-list> }
```

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) D can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by D? D might want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the base-list.

Note: Since a base class can itself be a derived class, the access attribute question is recursive: you backtrack until you reach the basemost of the base classes, those that do not inherit.

When declaring D, you can use the access specifier **public**, **protected**, or **private** in front of the classes in the base-list:

```
class D : public B1, private B2, ... {
  .
  .
  .
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they can alter the access attributes of base members as viewed by the derived class.

The default is private if D is a class declaration, and public if D is a struct declaration.

Note: Unions cannot have base classes, and unions cannot be used as base classes.

The derived class inherits access attributes from a base class as follows:

- **public** base class: **public** members of the base class are **public** members of the derived class. **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **protected** base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **private** base class: Both **public** and **protected** members of the base class are **private** members of the derived class. **private** members of the base class remain **private** to the base class.

Note that **private** members of a base class are always inaccessible to member functions of the derived class unless **friend** declarations are explicitly declared in the base class granting access. For example,

```
/* class X is derived from class A */
class X : A {               // default for class is private A
  .
  .
  .
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */
class Y : B, public C {     // override default for C
  .
  .
  .
}
/* struct S is derived from D */
struct S : D {              // default for struct is public D
  .
  .
  .
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
struct T : private D, E {   // override default for D
                            // E is public by default
  .
  .
```

```
  .
}
```

The effect of access specifiers in the base list can be adjusted by using a qualified-name in the public or protected declarations of the derived class. For example:

```
class B {
   int a;              // private by default
public:
   int b, c;
   int Bfunc(void);
};
class X : private B {   // a, b, c, Bfunc are now private in X
   int d;               // private by default, NOTE: a is not
                        // accessible in X
public:
   B::c;                // c was private, now is public
   int e;
   int Xfunc(void);
};
int Efunc(X& x);        // external to B and X
```

The function Efunc() can use only the public names c, e, and Xfunc().

The function Xfunc() is in X, which is derived from **private** B, so it has access to

- The “adjusted-to-public” c
- The “private-to-X” members from B: b and Bfunc()
- X’s own private and public members: d, e, and Xfunc()

However, Xfunc() cannot access the “private-to-B” member, a.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access

Virtual Base Classes (see page 116)

Friends Of Classes (see page 117)

1.1.1.3.8.12 Virtual Base Classes

This section contains Virtual Base Class topics.

1.1.1.3.8.12.1 Virtual Base Classes

A **virtual** class is a base class that is passed to more than one derived class, as might happen with multiple inheritance.

You cannot specify a base class more than once in a derived class:

```
class B { ... };
class D : B; B { ... }; //ILLEGAL
```

However, you can indirectly pass a base class to the derived class more than once:

```
class X : public B { ... };
class Y : public B { ... };
class Z : public X, public Y { ... }; //OK
```

In this case, each object of class Z has two sub-objects of class B.

If this causes problems, add the keyword **virtual** to the base class specifier. For example,

```
class X : virtual public B { ... };
class Y : virtual public B { ... };
class Z : public X, public Y { ... };
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes.

If the hierarchy contains multiple virtual base classes, the virtual base class constructors invoke in the order they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly constructed. For example, this code

```
class X : public Y , virtual public Z
 X one;
```

produces this order:

```
Z(); // virtual base class initialization
Y(); // non-virtual base class
X(); // derived class
```

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

1.1.1.3.8.13 Friends Of Classes

This section contains Friends Of Class topics.

1.1.1.3.8.13.1 Friends Of Classes

A **friend** F of a class X is a function or class, although not a member function of X, with full access rights to the private and protected members of X. In all other respects, F is a normal function with respect to scope, declarations, and definitions.

Since F is not a member of X, it is not in the scope of X, and it cannot be called with the x.F and xptr->F selector operators (where x is an X object and xptr is a pointer to an X object).

If the specifier **friend** is used with a function declaration or definition within the class X, it becomes a friend of X.

friend functions defined within a class obey the same inline rules as member functions (see Inline functions). **friend** functions are not affected by their position within the class or by any access specifiers. For example:

```
class X {
   int i;                          // private to X
   friend void friend_func(X*, int);
/* friend_func is not private, even though it's declared in the private section */
public:
     void member_func(int);
};
/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }
X xobj;
/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class Y into friends of class X with a single declaration:

```
class Y;                      // incomplete declaration
class X {
   friend Y;
   int i;
   void member_funcX();
};
class Y; {                    // complete the declaration
   void friend_X1(X&);
   void friend_X2(X*);
.
.
.
};
```

The functions declared in Y are friends of X, although they have no **friend** specifiers. They can access the private members of X, such as i and member_funcX.

It is also possible for an individual member function of class X to be a friend of class Y:

```
class X {
.
.
.
   void member_funcX();
}
class Y {
   int i;
   friend void X::member_funcX();
.
.
.
};
```

Class friendship is not transitive: X friend of Y and Y friend of Z does not imply X friend of Z. Friendship is not inherited.

See Also

C++ Classes (see page 100)

Class Names (see page 102)

Class Types (see page 103)

Class Name Scope (see page 103)

Class Objects (see page 104)

Class Member List (see page 105)

Member Functions (see page 106)

The Keyword This (see page 174)

Static Members (see page 106)

Inline Functions (see page 108)

Member Scope (see page 110)

Nested Types (see page 111)

Member Access Control (see page 112)

Base And Derived Class Access (see page 113)

Virtual Base Classes (see page 116)

Friends Of Classes

1.1.1.3.9 Constructors And Destructors

This section contains Constructor and Destructor topics.

1.1.1.3.9.1 Introduction To Constructors And Destructors

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features:

- They do not have return value declarations (not even **void**).

- They cannot be inherited, though a derived class can call the base class's constructors and destructors.
- Constructors, like most C++ functions, can have default arguments or use member initialization lists.
- Destructors can be **virtual**, but constructors cannot. (See Virtual destructors.)
- You can't take their addresses.

```
int main (void)
{
.
.
.
void *ptr = base::base;    // illegal
.
.
.
}
```

- Constructors and destructors can be generated by the compiler if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.
- You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
.
.
.
X *p;
.
.
.
p->X::~X();               // legal call of destructor
X::X();                   // illegal call of constructor
.
.
.
}
```

- The compiler automatically calls constructors and destructors when defining and destroying objects.
- Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
- An object with a constructor or destructor cannot be used as a member of a union.
- If no constructor has been defined for some class X to accept a given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class X. Note that this rule applies only to any constructor with one parameter and no initializers that use the "**=**" syntax.

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;                  // illegal: Y(X(1)) not tried
```

If **class** X has one or more constructors, one of them is invoked each time you define an object x of **class** X. The constructor creates x and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

1.1.1.3.9.2 Constructors

This section contains Constructor topics.

1.1.1.3.9.2.1 Constructors

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the main function is called. Global variable constructors are also called prior to **#pragma startup** functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {
public:
   X();   // class X constructor
};
```

A **class** X constructor cannot take X as an argument:

```
class X {
public:
   X(X);                      // illegal
};
```

The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the copy constructor. A constructor that accepts no parameters is called the default constructor.

1.1.1.3.9.2.2 Constructor Defaults

The default constructor for **class** X is one that takes no arguments; it usually has the form X::X(). If no user-defined constructors exist for a class, the compiler generates a default constructor. On a declaration such as X x, the default constructor creates the object x.

Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor X::X() takes no arguments and must not be confused with, say, X::X(int = 0), which can be called with no arguments as a default constructor, or can take an argument.

You should avoid ambiguity in defining constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
   X();
   X(int i = 0);
};
int main()
{
   X one(10);  // OK; uses X::X(int)
   X two;      // Error;ambiguous whether to call X::X() or
               // X::X(int = 0)
   return 0;
}
```

See Also

Constructors (see page 120)

1.1.1.3.9.2.3 The Copy Constructor

A copy constructor for **class** X is one that can be called with a single argument of type X as follows:

```
X::X(X&)
```

or

```
X::X(const X&)
```

or

```
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x1;
X x2 = x1;
X x3(x1);
```

The compiler generates a copy constructor for **class** X if one is needed and no other constructor has been defined in **class** X. The copy constructor that is generated by the compiler lets you safely start programming with simple data types. You need to make your own definition of the copy constructor if your program creates aggregate, complex types such as **class**, **struct**, and array types. The copy constructor is also called when you pass a class argument by value to a function.

See also the discussion of member-by-member class assignment. You should define the copy constructor if you overload the assignment operator.

See Also

Constructor Defaults (see page 120)

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

Example Of Overloading Operators (see page 129)

1.1.1.3.9.2.4 Overloading Constructors

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X {
   int    integer_part;
   double double_part;
public:
   X(int i)    { integer_part = i; }
   X(double d) { double_part = d; }
};
int main()
{
   X one(10);   // invokes X::X(int) and sets integer_part to 10
   X one(3.14); // invokes X::X(double) setting double_part to 3.14
```

```
    return 0;
}
```

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.2.5 Order Of Calling Constructors

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y();   // base class constructor
X();   // derived class constructor
```

For the case of multiple base classes,

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y();  // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any nonvirtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any nonvirtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a nonvirtual base, that nonvirtual base will be first so that the virtual base class can be properly constructed. The code:

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // nonvirtual base class
X();   // derived class
```

Or, for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base();      // virtual base class highest in hierarchy
             // base is constructed only once
base2();     // nonvirtual base of virtual base level2
             // must be called to construct level2
level2();    // virtual base class
base2();     // nonvirtual base of level1
level1();    // other nonvirtual base
toplevel();
```

If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.2.6 Class Initialization

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
   int i;
public:
   X();         // function bodies omitted for clarity
   X(int x);
   X(const X&);
};
void main()
{
   X one;        // default constructor invoked
   X two(1);     // constructor X::X(int) is used
   X three = 1;  // calls X::X(int)
   X four = one; // invokes X::X(const X&) for copy
   X five(two);  // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways:

1. It can accept the values as parameters and make assignments to the member variables within the function body of the

constructor:

```
class X
{
   int a, b;
public:
   X(int i, int j) { a = i; b = j }
};
```

2. An initializer list can be used prior to the function body:

```
class X
{
   int a, b, &c;  // Note the reference variable.
public:
   X(int i, int j) : a(i), b(j), c(a) {}
};
```

The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of X x(1, 2) assigns a value of 1 to x::a and 2 to x::b. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

Note: Base class constructors must be declared as either **public** or **protected** to be called from a derived class.

```
class base1
{
   int x;
public:
   base1(int i) { x = i; }
};
class base2
{
   int x;
public:
   base2(int i) : x(i) {}
};
class top : public base1, public base2
{
   int a, b;
public:
   top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j;}
};
```

With this class hierarchy, a declaration of top one(1, 2) would result in the initialization of base1 with the value 5 and base2 with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```
class X
{
  int a, b;
public:
  X(int i, j) :  a(i), b(a+j) {}
};
```

With this class, a declaration of X x(1,1) results in an assignment of 1 to x::a and 2 to x::b.

Base class constructors are called prior to the construction of any of the derived classes members. If the values of the derived class are changed, they will have no effect on the creation of the base class.

```
class base
{
   int x;
public:
   base(int i) : x(i) {}
```

```
};
class derived : base
{
   int a;
public:
   derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                         // passed an uninitialized 'a'
};
```

With this class setup, a call of derived d(1) will not result in a value of 10 for the base class member x. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
  .
  .
  .
}
```

See Also

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.3 Destructors

This section contains Destructor topics.

1.1.1.3.9.3.1 Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
   ~X(){};  // destructor for class X
};
```

If a destructor isn't explicitly defined for a class, the compiler generates one.

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.3.2 Invoking Destructors

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see Order of calling constructors).

See Also

Class Initialization (see page 123)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.3.3 atexit, #pragma exit, And Destructors

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the main function, are destroyed as they go out of scope. The order of execution at the end of a program is as follows:

- atexit() functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.3.4 Exit And Destructors

When you call exit from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Abort And Destructors (see page 127)

Virtual Destructors (see page 127)

1.1.1.3.9.3.5 abort And Destructors

If you call **abort** anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to **new**

```
#include <stdlib.h>
class X {
public:
   .
   .
   .
    ~X(){};
   .
   .
   .
};
void* operator new(size_t size, void *ptr)
{
   return ptr;
}
char buffer[sizeof(X)];
void main()
{
   X* pointer = new X;
   X* exact_pointer;
   exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer
.
.
.
   delete pointer;                 // delete used to destroy pointer
   exact_pointer->X::~X();         // direct call used to deallocate
}
```

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Virtual Destructors (see page 127)

1.1.1.3.9.3.6 Virtual Destructors

A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

```
/* How virtual affects the order of destructor calls.
   Without a virtual destructor in the base class, the derived
   class destructor won't be called. */
#include <iostream>
class color {
public:
   virtual ~color() {  // Virtual destructor
      std::cout << "color dtor\n";
      }
```

```
};
class red : public color {
public:
   ~red() {  // This destructor is also virtual
      std::cout << "red dtor\n";
      }
};
class brightred : public red {
public:
   ~brightred() {  // This destructor is also virtual
       std::cout << "brightred dtor\n";
       }
};
int main()
{
   color *palette[3];
   palette[0] = new red;
   palette[1] = new brightred;
   palette[2] = new color;
   // The destructors for red and color are called.
   delete palette[0];
   std::cout << std::endl;
   // The destructors for bright red, red, and color are called.
   delete palette[1];
   std::cout << std::endl;
   // The destructor for color is called.
   delete palette[2];
   return 0;
}
```

Program Output:

```
red dtor
color dtor
brightred dtor
red dtor
color dtor
color dtor
```

However, if no destructors are declared as virtual, **delete** palette[0], **delete** palette[1], and **delete** palette[2] would all call only the destructor for class color. This would incorrectly destruct the first two elements, which were actually of type red and brightred.

See Also

Class Initialization (see page 123)

Invoking Destructors (see page 126)

Atexit (see page 126)

Exit And Destructors (see page 126)

Abort And Destructors (see page 127)

1.1.1.3.10 Operator Overloading Overview

This section contains Operator Overloading Overview topics.

1.1.1.3.10.1 Overloading Operators

C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators can be overloaded except for:

```
.   .*   ::   ?:
```

The following preprocessing symbols cannot be overloaded.

```
#   ##
```

The =, [ ], ( ), and -> operators can be overloaded only as nonstatic member functions. These operators cannot be overloaded for **enum** types. Any attempt to overload a global version of these operators results in a compile-time error.

The keyword **operator** followed by the operator symbol is called the operator function name; it is used like a normal function name when defining the new (overloaded) action for the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function cannot alter the number of arguments or the precedence and associativity rules applying to normal operator use.

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.10.2 How To Construct A Class Of Complex Vectors

This section contains How to Construct A Class Of Complex Vector topics.

1.1.1.3.10.2.1 Example Of Overloading Operators

The following example extends the class complex to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in the usual mathematical syntax.

Some of the issues illustrated by the example are:

- The default constructor is defined. The default constructor is provided by the compiler only if you have not defined it or any other constructor.
- The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator.
- The assignment operator is overloaded. If you do not overload the assignment operator, the compiler calls a default assignment operator when required. By overloading assignment of cvector types, you specify exactly the actions to be taken. Note that derived classes cannot inherit the assignment operator.
- The subscript operator is defined as a member function (a requirement when overloading) with a single argument. The **const** version assures the caller that it will not modify its argument—this is useful when copying or assigning. This operator should check that the index value is within range—a good place to implement exception handling.
- The addition operator is defined as a member function. It allows addition only for cvector types. Addition should always check that the operands' sizes are compatible.
- The multiplication operator is declared a **friend**. This lets you define the order of the operands. An attempt to reverse the order of the operands is a compile-time error.

- The stream insertion operator is overloaded to naturally display a cvector. Large objects that don't display well on a limited size screen might require a different display strategy.

Example Source

```
/* HOW TO EXTEND THE complex CLASS AND OVERLOAD THE REQUIRED OPERATORS. */
complexcomplexcomplex
#include <complex>  // This includes iostream
using namespace std;
// COMPLEX VECTORS
template <class T>
class cvector {
   int size;
   complex<T> *data;
public:
   cvector() { size = 0; data = NULL; };
   cvector(int i = 5) : size(i) {    // DEFAULT VECTOR SIZE.
      data = new complex<T>[size];
      for (int j = 0; j < size; j++)
           data[j] = j + (0.1 * j);  // ARBITRARY INITIALIZATION.
      };
   /* THIS VERSION IS CALLED IN main() */
   complex<T>& operator [](int i) { return data[i]; };
   /* THIS VERSION IS CALLED IN ASSIGNMENT OPERATOR AND COPY THE CONSTRUCTOR */
   const complex<T>& operator [](int i) const { return data[i]; };
   cvector operator +(cvector& A) {  // ADDITION OPERATOR
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < size; i++)
          result[i] = data[i] + A.data[i];
      return result;
      };
   /* BECAUSE scalar * vector MULTIPLICATION IS NOT COMMUTATIVE, THE ORDER OF
      THE ELEMENTS MUST BE SPECIFIED. THIS FRIEND OPERATOR FUNCTION WILL ENSURE
      PROPER MULTIPLICATION. */
   friend cvector operator *(T scalar, cvector& A) {
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < A.size; i++)
        result.data[i] = scalar * A.data[i];
      return result;
      }
   /* THE STREAM INSERTION OPERATOR. */
   friend ostream& operator <<(ostream& out_data, cvector& C) {
      for (int i = 0; i < C.size; i++)
           out_data << "[" << i << "]=" << C.data[i] << "   ";
      cout << endl;
      return out_data;
      };
   cvector( const cvector &C ) {  // COPY CONSTRUCTOR
      size = C.size;
      data = new complex<T>[size];
      for (int i = 0; i < size; i++)
           data[i] = C[i];
     }
   cvector& operator =(const cvector &C) { // ASSIGNMENT OPERATOR.
      if (this == &C) return *this;
      delete[] data;
      size = C.size;
      data = new complex<T>[size];
      for (int i = 0; i < size; i++)
          data[i] = C[i];
      return *this;
   };
   virtual ~cvector() { delete[] data; }; // DESTRUCTOR
   };
int main(void) { /* A FEW OPERATIONS WITH complex VECTORS. */
   cvector<float> cvector1(4), cvector2(4), result(4);
   // CREATE complex NUMBERS AND ASSIGN THEM TO complex VECTORS
```

```
    cvector1[3] = complex<float>(3.3, 102.8);
    cout << "Here is cvector1:" << endl;
    cout << cvector1;
    cvector2[3] = complex<float>(33.3, 81);
    cout << "Here is cvector2:" << endl;
    cout << cvector2;
    result = cvector1 + cvector2;
    cout << "The result of vector addition:" << endl;
    cout << result;
    result = 10 * cvector2;
    cout << "The result of 10 * cvector2:" << endl;
    cout << result;
    return 0;
    }
```

Output

```
Here is cvector1:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(3.3, 102.8)
Here is cvector2:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(33.3, 81)
The result of vector addition:
[0]=(0, 0)   [1]=(2.2, 0)   [2]=(4.4, 0)   [3]=(36.6, 183.8)
The result of 10 * cvector2:
[0]=(0, 0)   [1]=(11, 0)   [2]=(22, 0)   [3]=(333, 810)
```

1.1.1.3.11 Overloading Operator Functions Overview

This section contains Overloading Operator Functions Overview topics.

1.1.1.3.11.1 Overloading Operator Functions

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2);   // same as c3 = c1 + c2
```

Apart from new and delete, which have their own rules, an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions =, ( ), [ ] and -> must be nonstatic member functions.

Enumerations can have overloaded operators. However, the operator functions =, ( ), [ ], and -> cannot be overloaded for enumerations.

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading The Assignment Operator = (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.11.2 Overloaded Operators And Inheritance

With the exception of the assignment function operator =( ), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could possibly be further overloaded for Y.

See Also

Example Of Overloading Operators (see page 129)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.11.3 Overloading Unary Operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either x.operator@() or operator@(x), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

- Under C++ 2.0, an overloaded operator ++ or -- is used for both prefix and postfix uses of the operator.
- With C++ 2.1, when an operator ++ or operator -- is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix operator ++ or operator --. You can only overload a postfix operator ++ or operator -- by defining it as a member function taking an int parameter or as a nonmember function taking one class and one int parameter.

When only the prefix version of an operator ++ or operator -- is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator in function func()
Warning: Overloaded prefix 'operator --' used as a postfix operator in function func()
```

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.11.4 Overloading Binary Operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually friend) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either x.operator@(y) or operator@(x,y) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.11.5 Overloading The Assignment operator =

The assignment operator=( ) can be overloaded by declaring a nonstatic member function. For example,

```
class String {
      .
      .
      .
   String& operator = (String& str);
      .
      .
      .
   String (String&);
   ~String();
}
```

This code, with suitable definitions of String::operator =(), allows string assignments str1 = str2 in the usual sense. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class X, there is no user-defined operator =, the operator = is defined by default as a member-by-member assignment of the members of class X:

```
X& X::operator = (const X& source)
{
   // memberwise assignment
}
```

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.11.6 Overloading The Function Call Operator ( )

Syntax

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

Description

In its ordinary use as a function call, the postfix-expression must be a function name, or a pointer or reference to a function.

When the postfix-expression is used to make a member function call, postfix-expression must be a class member function name or a pointer-to-member expression used to select a class member function. In either case, the postfix-expression is followed by the optional expression-list.

A call X(arg1, arg2), where X is an object class X, is interpreted as X.**operator()**(arg1, arg2).

The function call operator, **operator()()**, can only be overloaded as a nonstatic member function.

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.11.7 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.

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Class Member Access Operators -> (see page 134)

1.1.1.3.11.8 Overloading The Class Member Access Operators ->

Syntax

```
postfix-expression -> primary-expression
```

Description

The expression x->m, where x is a **class** X object, is interpreted as (x.operator->())->m, so that the function **operator->()** must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

The **operator->()** can only be overloaded as a nonstatic member function.

See Also

Example Of Overloading Operators (see page 129)

Overloaded Operators And Inheritance (see page 131)

Overloading Binary Operators (see page 132)

Overloading Operator Functions (see page 131)

Overloading The Assignment Operator = (see page 133)

Overloading The Function Call Operator ( ) (see page 133)

Overloading The Subscript Operator [] (see page 134)

1.1.1.3.12 Polymorphic Classes

This section contains Polymorphic Class topics.

1.1.1.3.12.1 Polymorphic Classes

Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are **class** and **struct**.

See Also

Virtual Functions (see page 135)

Dynamic Functions (see page 137)

Abstract Classes (see page 139)

1.1.1.3.12.2 Virtual Functions

This section contains Virtual Function topics.

1.1.1.3.12.2.1 Virtual Functions

virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the **virtual** keyword. To declare a pure function (which automatically declares an abstract class), prefix the prototype with the **virtual** keyword, and set the function equal to zero.

```
virtual int funct1(void);       // A virtual function declaration.
virtual int funct2(void) = 0;   // A pure function declaration.
```

A function declaration cannot provide both a pure-specifier and a definition.

Example

```
struct C {
    virtual void f() { } = 0; // ill-formed
};
```

The only legal syntax to provide a body is:

```
struct TheClass
{
  virtual void funct3(void) = 0;
};
```

```
virtual void TheClass::funct3(void)
{
   // Some code here.
};
```

Note: See Abstract classesAbstractClasses for a discussion of pure virtual functions.

When you declare **virtual** functions, keep these guidelines in mind:

- They can be member functions only.
- They can be declared a **friend** of another class.
- They cannot be a static member.

A **virtual** function does not need to be redefined in a derived class. You can supply one definition in the base class so that all calls will access the base function.

To redefine a **virtual** function in any derived class, the number and type of arguments must be the same in the base class declaration and in the derived class declaration. (The case for redefined **virtual** functions differing only in return type is discussed below.) A redefined function is said to override the base class function.

You can also declare the functions int Base::Fun(int) and int Derived::Fun(int) even when they are not virtual. In such a case, int Derived::Fun(int) is said to hide any other versions of Fun(int) that exist in any base classes. In addition, if class Derived defines other versions of Fun(), (that is, versions of Fun() with different signatures) such versions are said to be overloaded versions of Fun().

Virtual function return types

Generally, when redefining a **virtual** function, you cannot change just the function return type. To redefine a **virtual** function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the **virtual** function mechanism is ignored.

However, for certain virtual functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when both of the following conditions are met:

- The overridden **virtual** function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class B and class D (derived publicly from B) each contain a **virtual** function vf, then if vf is called for an object d of D, the call made is D::vf(), even when the access is via a pointer or reference to B. For example,

```
struct X {};// Base class.
struct Y : X {};// Derived class.
struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
   virtual X* pf();// Return type is a pointer to base. This can
  //  be overridden.
};
class D : public B {
public:
   virtual void vf1();// Virtual specifier is legal but redundant.
   void vf2(int);// Not virtual, since it's using a different
 //  arg list. This hides B::vf2().
// char vf3();// Illegal: return-type-only change!
   void f();
   Y*   pf();// Overriding function differs only
   //  in return type. Returns a pointer to
 //  the derived class.
```

```
};
void extf()
{
   D d;// Instantiate D
   B* bp = &d;// Standard conversion from D* to B*
  // Initialize bp with the table of functions
// provided for object d. If there is no entry for a
  // function in the d-table, use the function
     //  in the B-table.
   bp->vf1();  // Calls D::vf1
   bp->vf2();  // Calls B::vf2 since D's vf2 has different args
   bp->f();    // Calls B::f (not virtual)
   X* xptr = bp->pf();// Calls D::pf() and converts the result
   //  to a pointer to X.
   D* dptr = &d;
   Y* yptr = dptr->pf();// Calls D::pf() and initializes yptr.
   //  No further conversion is done.
}
```

The overriding function vf1 in D is automatically **virtual**. The **virtual** specifier can be used with an overriding function declaration in the derived class. If other classes will be derived from D, the **virtual** keyword is required. If no further classes will be derived from D, the use of **virtual** is redundant.

The interpretation of a **virtual** function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at runtime (late binding).

See Also

Polymorphic Classes (see page 135)

Dynamic Functions (see page 137)

Abstract Classes (see page 139)

1.1.1.3.12.3 Dynamic Functions

dynamic functions are allowed for classes derived from TObject. Dynamic functions are similar to virtual functions except for the way they are stored in the virtual tables. Virtual functions occupy a slot in the virtual table in the object they are defined in, and in the virtual table of every descendant of that object. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of its ancestors are searched until the function is found.

Therefore, dynamic functions reduces the size of your virtual tables at the expense of a delay at runtime to look up the address of the functions.

Because dynamic functions are available only in classes derived from TObject, you will get an error if you use them in a regular class. For example:

```
class dynfunc {
int __declspec(dynamic) bar() { return 5; }
};
```

gives the syntax error, “Error: Storage class 'dynamic' is not allowed here.” But, the following code compiles.

```
#include <clxvcl.h>
#include <stdio.h>
class __declspec(delphiclass) func1 : public TObject {
public:
func1() {}
int virtual virtbar() { return 5; }
int __declspec(dynamic) dynbar() { return 5; }
```

```
};
class __declspec(delphiclass) func2 : public func1 {
public:
func2() {}
};
class __declspec(delphiclass) func3 : public func2 {
public:
func3() {}
int virtbar() { return 10; }
int dynbar() { return 10; }
};
int main()
{
func3 * Func3 = new func3;
func1 * Func1 = Func3;
printf("func3->dynbar: %d\n", Func3->dynbar());
printf("func3->virtbar: %d\n", Func3->virtbar());
printf("func1->dynbar: %d\n", Func1->dynbar());
printf("func1->virtbar: %d\n", Func1->virtbar());
delete Func3;
func2 * Func2 = new func2;
printf("func2->dynbar: %d\n", Func2->dynbar());
printf("func2->virtbar: %d\n", Func2->virtbar());
delete Func2;
return 0;
}
```

Dynamic attribute is inherited

Since dynamic functions are just like virtual functions, the dynamic attribute is automatically inherited. You can verify this by running the above example. When you generate assembly output with "bcc32 -S" you can examine the virtual tables of func1, func2, and func3, and you'll see how func2 has NO entry for dynbar, but it does have an entry for virtbar. Still, you can call dynbar in the func2 object:

Output:

```
func3->dynbar: 10
func3->virtbar: 10
func1->dynbar: 10
func1->virtbar: 10
func2->dynbar: 5
func2->virtbar: 5
```

Dynamic functions cannot be made virtual, and vice-versa

You cannot redeclare a virtual function to be dynamic; likewise, you cannot redeclare a dynamic function to be virtual. The following example gives errors:

```
#include <clxvcl.h>
#include <stdio.h>
class __declspec(delphiclass) foo1 : public TObject {
public:
foo1() {}
int virtual virtbar() { return 5; }
int __declspec(dynamic) dynbar() { return 5; }
};
class __declspec(delphiclass) foo2 : public foo1 {
public:
foo2() {}
int __declspec(dynamic) virtbar() { return 10; }
int virtual dynbar() { return 10; }
};
Error : Cannot override a virtual with a dynamic function
Error : Cannot override a dynamic with a virtual function
```

See Also

Polymorphic Classes (see page 135)

Virtual Functions (see page 135)

Dynamic Functions

Abstract Classes (see page 139)

1.1.1.3.12.4 Abstract Classes

This section contains Abstract Class topics.

1.1.1.3.12.4.1 Abstract Classes

An abstract class is a class with at least one pure **virtual** function. A **virtual** function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {       // abstract class
   point center;
.
.
.
public:
   where() { return center; }
   move(point p) { center = p; draw(); }
   virtual void rotate(int) = 0; // pure virtual function
   virtual void draw() = 0;      // pure virtual function
   virtual void hilite() = 0;    // pure virtual function
.
.
.
}
shape x;// ERROR: attempt to create an object of an abstract class
   shape* sptr;// pointer to abstract class is OK
   shape f();// ERROR: abstract class cannot be a return type
int g(shape s);// ERROR: abstract class cannot be a function argument type
shape& h(shape&);// reference to abstract class as return
 // value or function argument is OK
```

Suppose that D is a derived class with the abstract class B as its immediate base class. Then for each pure virtual function pvf in B, if D doesn't provide a definition for pvf, pvf becomes a pure member function of D, and D will also be an abstract class.

For example, using the class shape previously outlined,

```
class circle : public shape {// circle derived from abstract class
   int radius;// private
public:
   void rotate(int) { }// virtual function defined: no action
            //  to rotate a circle
   void draw();      // circle::draw must be defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a runtime error.

See Also

Polymorphic Classes (see page 135)

Virtual Functions (see page 135)

Dynamic Functions (see page 137)

1.1.1.3.13 C++ Scope

This section contains C++ Scope topics.

1.1.1.3.13.1 C++ Scope

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

See Also

Class Scope (see page 140)

Hiding (see page 140)

C++ Scoping Rules Summary (see page 141)

1.1.1.3.13.2 Class Scope

The name M of a member of a class X has class scope “local to X”; it can be used only in the following situations:

- In member functions of X
- In expressions such as x.M, where x is an object of X
- In expressions such as xptr->M, where xptr is a pointer to an object of X
- In expressions such as X::M or D::M, where D is a derived class of X
- In forward references within the class of which it is a member

Names of functions declared as friends of X are not members of X; their names simply have enclosing scope.

See Also

C++ Scope (see page 140)

Hiding (see page 140)

C++ Scoping Rules Summary (see page 141)

1.1.1.3.13.3 Hiding

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: X::M. A hidden file scope (global) name can be referenced with the unary operator :: (for example, ::g). A class name X can be hidden by the name of an object, function, or enumerator declared within the scope of X, regardless of the order in which the names are declared. However, the hidden class name X can still be accessed by prefixing X with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name x is immediately after its complete declaration but before its initializer, if one exists.

See Also

C++ Scope (see page 140)

Class Scope (see page 140)

C++ Scoping Rules Summary (see page 141)

1.1.1.3.13.4 C++ Scoping Rules Summary

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

- The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
- If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
- If the name is used outside any function and class, or is prefixed by the unary scope access operator ::, and if the name is not qualified by the binary :: operator or the member selection operators **.** and **->**, then the name must be a global object, function, or enumerator.
- If the name n appears in any of the forms X::n, x.n (where x is an object of X or a reference to X), or ptr->n (where ptr is a pointer to X), then n is the name of a member of X or the member of a class from which X is derived.
- Any name that hasn't been discussed yet and that is used in a static member function must either be declared in the block it occurs in or in an enclosing block, or be a global name. The declaration of a local name n hides declarations of n in enclosing blocks and global declarations of n. Names in different scopes are not overloaded.
- Any name that hasn't been discussed yet and that is used in a nonstatic member function of class X must either be declared in the block it occurs in or in an enclosing block, be a member of class X or a base class of X, or be a global name. The declaration of a local name n hides declarations of n in enclosing blocks, members of the function's class, and global declarations of n. The declaration of a member name hides declarations of the same name in base classes.
- The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a nondefining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
- A constructor initializer is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

See Also

C++ Scope (see page 140)

Class Scope (see page 140)

Hiding (see page 140)

1.1.1.3.14 Templates

This section contains Template topics.

1.1.1.3.14.1 Using Templates

Templates, also called generics or parameterized types, let you construct a family of related functions or classes. These topics introduce the basic concept of templates:

Exporting and importing templates (see page 147)

Template Body Parsing (see page 142)

Function Templates (see page 143)

Class Templates (see page 144)

Implicit and Explicit Template Functions (see page 144)

Template Compiler Switches (see page 146)

1.1.1.3.14.2 template

Category

C++-Specific Keywords

Syntax

```
template-declaration:templateclass
   template < template-argument-list > declaration
template-argument-list:
   template-argument
   template-argument-list, template argument
template-argument:
   type-argument
   argument-declaration
type-argument:
   class typename identifier
template-class-name:
   template-name < template-arg-list >
template-arg-list:
   template-arg
   template-arg-list , template-arg
template-arg:
   expression
   type-name
< template-argument-list > declaration
```

Description

Use templates (also called generics or parameterized types) to construct a family of related functions or classes.

1.1.1.3.14.3 Template Body Parsing

Earlier versions of the compiler didn't check the syntax of a template body unless the template was instantiated. A template body is now parsed immediately when seen like every other declaration.

```
template <class T> class X : T
{
  Int  j;  // Error: Type name expected in template X<T>
};
```

Let's assume that Int hasn't yet been defined. This means that Int must be a member of the template argument T. But it also might just be a typing error and should be **int** instead of Int. Because the compiler can't guess the right meaning it issues an error message.

If you want to access types defined by a template argument you should use a **typedef** to make your intention clear to the compiler:

```
template <class T> class X : T
{
  typedef  typename T::Int  Int;
  Int  j;
};
```

You cannot just write

```
   typedef   T::Int;
```

as in earlier versions of the compiler. Not giving the **typedef** name was acceptable, but this now causes an error message.

All other templates mentioned inside the template body are declared or defined at that point. Therefore, the following example is ill-formed and will not compile:

```
template <class T> class  X
```

```
{
  void f(NotYetDefindedTemplate<T> x);
};
```

All template definitions must end with a semicolon. Earlier versions of the compiler did not complain if the semicolon was missing.

See Also

Using Templates (see page 141)

Function Templates (see page 143)

Class Templates (see page 144)

Template Compiler Switches (see page 146)

Template Generation Semantics (see page 147)

Exporting And Importing Templates (see page 147)

1.1.1.3.15 Function Templates Overview

This section contains Function Templates Overview topics.

1.1.1.3.15.1 Function Templates

Consider a function max(x, y) that returns the larger of its two arguments. x and y can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters x and y to be declared at compile time. Without using templates, many overloaded versions of max are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the #**define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of max(x, y) is to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be. By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

```
template <class T> T max(T x, T y){
   return (x > y) ? x : y;
   };
```

The data type is represented by the template argument **<class T>**. When used in an application, the compiler generates the appropriate code for the max function according to the data type actually used in the call:

```
int i;
Myclass a, b;
int j = max(i,0);       // arguments are integers
Myclass m = max(a,b);   // arguments are type Myclass
```

Any data type (not just a class) can be used for **<class T>**. The compiler takes care of calling the appropriate **operator>()**, so you can use max with arguments of any type for which **operator>()** is defined.

See Also

Exporting And Importing Templates (see page 147)

Implicit And Explicit Template Functions (see page 144)

1.1.1.3.15.2 Overriding A Template Function

The previous example is called a function template (or generic function, if you like). A specific instantiation of a function template is called a template function. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>
char *max(char *x, char *y){
   return(strcmp(x,y) > 0) ? x : y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

1.1.1.3.15.3 Implicit And Explicit Template Functions

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b){
        return  (a > b) ? a : b;
}
void f(int i, char c)
{
    max(i, i);     // calls max(int ,int )
    max(c, c);     // calls max(char,char)
    max(i, c);     // no match for max(int,char)
    max(c, i);     // no match for max(char,int)
}
```

This code results in the following error messages:

```
Could not find a match for 'max(int,char)' in function f(int,char)Could not find a match for
'max(char,int)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)
```

If the user explicitly declares a function, this function, on the other hand, will participate fully in overload resolution. See the example of explicit function.

When searching for an exact match for template function parameters, trivial conversions are considered to be exact matches. See the example on trivial conversions.

Template functions with derived class pointer or reference arguments are permitted to match their public base classes. See the example of base class referencing.

1.1.1.3.16 Class Templates Overview

This section contains Class Templates Overview topics.

1.1.1.3.16.1 Class Templates

A class template (also called a generic class or class generator) lets you define a pattern for class definitions. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a type

parameter to the class, the system will generate type-safe class definitions on the fly.

As with function templates, an explicit template class specialization can be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol Vector must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

See Also

Exporting And Importing Templates (see page 147)

Eliminating Pointers In Templates (see page 146)

Template Arguments (see page 145)

1.1.1.3.16.2 Template Arguments

Multiple arguments are allowed as part of the class template declaration. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Both non-type template arguments such as size and type arguments can have default values. The value supplied for a non-type template argument must be a constant expression:

```
const int N = 128;
int i = 256;
Buffer<int, 2*N> b1;// OK
Buffer<float, i> b2;// Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

1.1.1.3.16.3 Using Angle Brackets In Templates

Be careful when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the > between x and 100 would prematurely close the template argument list.

1.1.1.3.16.4 Using Type-safe Generic Lists In Templates

In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

```
class GList
{
 public:
   void insert( void * );
   void *peek();
    .
    .
    .
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public GList {
```

```
 public:
   void insert( Foo *f ) { GList::insert( f ); }
   Foo *peek() { return (Foo *)GList::peek(); }
       .
       .
       .
};
```

This is type-safe. insert will only take arguments of type pointer-to-Foo or object-derived-from-Foo, so the underlying container will only hold pointers that in fact point to something of type Foo. This means that the cast in FooList::peek() is always safe, and you've created a true FooList. Now, to do the same thing for a BarList, a BazList, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and type-safety, you can once again use templates. See the example for type-safe generic list class. The C++ Standard Template Library (STL) has a rich set of type-safe collection classes.

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no runtime overhead imposed by this type safety.

1.1.1.3.16.5 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.

1.1.1.3.17 Compiler Template Switches

This section contains Compiler Template Switch topics.

1.1.1.3.17.1 Template Compiler Switches

The -Jg family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler.

For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which instances will be user-defined so that they aren't generated from the template.

1.1.1.3.18 Template Generation Semantics

The C++ compiler generates the following methods for template instances:

- Those methods which were actually used
- Virtual methods of an instance
- All methods of explicitly instantiated classes

The advantage of this behavior is that it results in significantly smaller object files, libraries and executable files, depending on how heavily you use templates.

Optionally, you can use the '-Ja' switch to generate all methods.

You can also force all of the out-of-line methods of a template instance to be generated by using the explicit template instantiation syntax defined in the ISO/ANSI C++ Standard. The syntax is:

template class classname**<**template parameter**>;**

The following STL example directs the compiler to generate all out-of-line methods for the "list<char>" class, regardless of whether they are referenced by the user's code:

```
template class list<char>
```

You can also explicitly instantiate a single method, or a single static data member of a template class, which means that the method is generated to the .OBJ even though it is not used:

```
template void classname <template parameter>:: methodname ();
```

1.1.1.3.19 Exporting And Importing Templates

This section contains Exporting And Importing Template topics.

1.1.1.3.19.1 Exporting And Importing Templates

The declaration of a template function or template class needs to be sufficiently flexible to allow it to be used in either a dynamic link library (shared library) or an executable file. The same template declaration should be available as an import and/or export, or without a modifier. To be completely flexible, the header file template declarations should not use __**export** or __**import** modifiers. This allows the user to apply the appropriate modifier at the point of instantiation depending on how the instantiation is to be used.

The following steps demonstrate exporting and importing of templates. The source code is organized in three files. Using the header file, code is generated in the dynamic link library.

1. Exportable/Importable Template Declarations

The header file contains all template class and template function declarations. An export/import version of the templates can be instantiated by defining the appropriate macro at compile time.

2. Compiling Exportable Templates

Write the source code for a dynamic link library. When compiled, this library has reusable export code for templates.

3. Using ImportTemplates

Now you can write a calling function that uses templates. This executable file is linked to the dynamic link library. Only objects

that are not declared in the header file and which are instantiated in the main function cause the compiler to generate new code. Code for a newly instantiated object is written into main.obj file.

1.1.1.4 The Preprocessor

This section contains Preprocessor topics.

1.1.1.4.1 Preprocessor Directives

This section contains Preprocessor Directive topics.

1.1.1.4.1.1 Preprocessor Directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them. The preprocessor supports these directives:

# (null directive)	**#ifdef**
#define	**#ifndef**
#elif	**#undef**
#else	**#include**
#endif	**#line**
#error	**#pragma**
#if	**#import**

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

1.1.1.4.1.2 # (null Directive)

Syntax

#

Description

The null directive consists of a line containing the single character #. This line is always ignored.

1.1.1.4.2 Defining And Undefining Macros

This section contains Defining And Undefining Macro topics.

1.1.1.4.2.1 #define

Syntax

#define macro_identifier <token_sequence>

Description

The **#define** directive defines a macro. Macros provide a mechanism for token replacement with or without a set of formal,

function-like parameters.

Each occurrence of **macro_identifier** in your source code following this control line will be replaced in the original position with the possibly empty **token_sequence** (there are some exceptions, which are noted later). Such replacements are known as macro expansions. The token sequence is sometimes called the body of the macro.

An empty token sequence results in the removal of each affected macro identifier from the source code.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of nested macros: The expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, the directive will not be recognized by the preprocessor. There are these restrictions to macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro won't be expanded during its own expansion (so **#define** A A won't expand indefinitely).

Example

```
#define HI "Have a nice day!"
#define empty
```

See Also

Keywords And Protected Words In Macros (see page 150)

Macros With Parameters (see page 150)

#undef (see page 149)

Using The -D And -U Command-line Options (see page 150)

1.1.1.4.2.2 #undef

Syntax

```
#undef macro_identifier
```

Description

You can undefine a macro using the **#undef** directive. **#undef** detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined. No macro expansion occurs within **#undef** lines.

The state of being defined or undefined turns out to be an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

See Also

#define (see page 148)

Preprocessor Directives (see page 148)

1.1.1.4.2.3 Using The -D And -U Command-line Options

Identifiers can be defined and undefined using the command-line compiler options -D and -U.

The command line

```
BCC32 -Ddebug=1; paradox=0; X -Umysym myprog.cbc++ -Ddebug=1:paradox=0:X -Umysym myprog.c
bc++ -Ddebug=1:paradox=0:X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
```

in the program.

See Also

#define (see page 148)

1.1.1.4.2.4 Keywords And Protected Words In Macros

It is legal but ill-advised to use C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long    /* legal and possibly useful */
```

The following predefined global identifiers cannot appear immediately following a **#define** or **#undef** directive:

- **__DATE__ __FILE__ __LINE__**
- **__STDC__ __TIME__**

1.1.1.4.3 Macros With Parameters Overview

This section contains Macros With Parameters Overview topics.

1.1.1.4.3.1 Macros With Parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note there can be no whitespace between the macro identifier and the **(**. The optional arg_list is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a formal argument or placeholder.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional actual_arg_list must contain the same number of comma-delimited token sequences, known as actual arguments,

as found in the formal arg_list of the **#define** line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the actual_arg_list.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

See Also

Converting To Strings With # (see page 151)

Nesting Parentheses And Commas (see page 151)

Side Effects And Other Dangers (see page 152)

Using The Backslash (\) For Line Continuation (see page 152)

1.1.1.4.3.2 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");
/* showerr("Error",2,"Press Enter, then ESC"); */
return SUM(f(i, j), g(k, l));
/* return ( (f(i, j)) + (g(k, l)) ); */
```

1.1.1.4.3.3 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).

1.1.1.4.3.4 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);
```

1.1.1.4.3.5 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."
```

1.1.1.4.3.6 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;
```

1.1.1.4.4 File Inclusion With #include

This section contains File Inclusion With #include topics.

1.1.1.4.4.1 #include

Syntax

```
#include <header_name>
#include "header_name"
```

Description

The **#include** directive pulls in other named files, known as include files, header files, or headers, into the source code. The syntax has three versions:

- The first and second versions imply that no macro expansion will be attempted; in other words, header_name is never scanned for macro identifiers. header_name must be a valid file name with an extension (traditionally .h for header) and optional path name and path delimiters.
- The third version assumes that neither **<** nor " appears as the first non-whitespace character following **#include**; further, it assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <header_name> or "header_name" formats.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the header_name, only that directory will be searched.

The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file.

1.1.1.4.4.2 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.

1.1.1.4.4.3 Header File Search With "header_name"

The "header_name" version specifies a user-supplied include file; the file is searched in the following order:

1. 1. - in the same directory of the file that contains the **#include** statement
2. 2. - in the directories of files that include (**#include**) that file
3. 3. - the current directory
4. 4. - along the path specified by the /I compiler option

1.1.1.4.5 Conditional Compilation Overview

This section contains Conditional Compilation Overview topics.

1.1.1.4.5.1 Conditional Compilation

The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

1.1.1.4.5.2 Defined

Syntax

```
#if defined[(] <identifier> [)]
#elif defined[(] <identifier> [)]
```

Description

Use the **defined** operator to test if an identifier was previously defined using **#define**. The **defined** operator is only valid in **#if** and **#elif** expressions.

Defined evaluates to 1 (true) if a previously defined symbol has not been undefined (using **#undef**); otherwise, it evaluates to 0 (false).

Defined performs the same function as **#ifdef**.

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use defined repeatedly in a complex expression following the **#if** directive; for example,

```
#if defined(mysym) && !defined(yoursym)
```

See Also

#ifdef And #ifndef (see page 154)

1.1.1.4.5.3 #if, #elif, #else, And #endif

Syntax

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
  .
```

```
   .
   .
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
```

Description

The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. If the constant-expression-1 (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by section-1, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the compiler. Otherwise, if constant-expression-1 evaluates to zero (false), section-1 is ignored (no macro expansion and no compilation).

In the true case, after section-1 has been preprocessed, control passes to the matching **#endif** (which ends this conditional sequence) and continues with next-section. In the false case, control passes to the next **#elif** line (if any) where constant-expression-2 is evaluated. If true, section-2 is processed, after which control moves on to the matching **#endif**. Otherwise, if constant-expression-2 is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be matched with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

See Also

Defined (see page 153)

#ifdef And #ifndef (see page 154)

1.1.1.4.5.4 #ifdef And #ifndef

Syntax

```
#ifdef identifier
#ifndef identifier
```

Description

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous **#define** command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if identifier is currently defined, and the same effect as

```
#if 0
```

if identifier is currently undefined.

#ifndef tests true for the "not-defined" condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if identifier is currently defined, and the same effect as

```
#if 1
```

if identifier is currently undefined.

The syntax thereafter follows that of the **#if**, **#elif**, **#else**, and **#endif**.

An identifier defined as NULL is considered to be defined.

1.1.1.4.6 The #line Control Directive

This section contains #line Control Directive topics.

1.1.1.4.6.1 #line

Syntax

```
#line integer_constant  <"filename">
```

Description

You can use the **#line** directive to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program.

The **#line** directive indicates that the following source line originally came from line number integer_constant of filename. Once the filename has been registered, subsequent **#line** commands relating to that file can omit the explicit filename argument.

Macros are expanded in **#line** arguments as they are in the **#include** directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

1.1.1.4.7 The #error Control Directive

This section contains #error Control Directive topics.

1.1.1.4.7.1 #error

Syntax

```
#error errmsg
```

Description

The **#error** directive generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional statement that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional statement that is true for the undesired case.

1.1.1.4.8 Pragma Directives Overview

This section contains Pragma Directives Overview topics.

1.1.1.4.8.1 #pragma alias

Syntax

```
#pragma alias "name1"="name2"
```

Description

Use **#pragma alias** to tell the linker that two identifier names are equivalent. You must put the two names in double quotes separated by an equal sign.

1.1.1.4.8.2 #pragma

Syntax

```
#pragma directive-name
```

Description

With **#pragma**, you can set compiler directives in your source code, without interfering with other compilers that also support **#pragma**. If the compiler doesn't recognize directive-name, it ignores the **#pragma** directive without any error or warning message.

The CodeGear C++ supports the following **#pragma** directives: (Llinks to these topics are at the bottom of this topic.)

- **#pragma alignment**
- **#pragma anon_struct**
- **#pragma argsused**
- **#pragma checkoption**
- **#pragma codeseg**
- **#pragma comment**
- **#pragma defineonoption**
- **#pragma exit**
- **#pragma hdrfile**
- **#pragma hdrstop**
- **#pragma inline**
- **#pragma intrinsic**
- **#pragma link**
- **#pragma message**
- **#pragma nopushoptwarn**
- **#pragma obsolete**
- **#pragma option**
- **#pragma pack**
- **#pragma package**
-

#pragma resource

- **#pragma startup**
- **#pragma undefineonoption**
- **#pragma warn**

See Also

#pragma alignment (see page 157)

#pragma anon_struct (see page 158)

#pragma argsused (see page 158)

#pragma checkoption (see page 158)

#pragma codeseg (see page 158)

#pragma comment (see page 159)

#pragma curious_george (see page 160)

#pragma defineonoption (see page 160)

#pragma exit (see page 159)

#pragma hdrfile (see page 159)

#pragma hdrstop (see page 160)

#pragma inline (see page 161)

#pragma intrinsic (see page 161)

#pragma link (see page 161)

#pragma message (see page 162)

#pragma nopushoptwarn (see page 162)

#pragma obsolete (see page 165)

#pragma option (see page 165)

#pragma pack (see page 162)

#pragma package (see page 164)

#pragma resource (see page 167)

#pragma startup (see page 159)

#pragma undefineonoption (see page 160)

#pragma warn (see page 167)

1.1.1.4.8.3 #pragma alignment

Syntax

```
#pragma alignment
```

Description

The **alignment** directive emits a message specifying the current alignment and enum size. For example:

```
W8035: The alignment is 8 bytes, the enum size is 4 bytes
```

The message will only be displayed in the IDE if the Compiler options for"Show general messages is checked. The message can also be enabled or disabled using the compiler's **-wmsg** switch.

1.1.1.4.8.4 #pragma anon_struct

Syntax

```
#pragma anon_struct on
#pragma anon_struct off
```

Description

The **anon_struct** directive allows you to compile anonymous structures embedded in classes.

```
#pragma anon_struct on
struct S {
   int i;
   struct {  // Embedded anonymous struct
      int   j ;
      float x ;
   };
   class {  // Embedded anonymous class
   public:
      long double ld;
   };
   S() { i = 1; j = 2; x = 3.3; ld = 12345.5;}
};
#pragma anon_struct off
void main()
{
   S mystruct;
   mystruct.x = 1.2;  // Assign to embedded data.
}
```

1.1.1.4.8.5 #pragma argsused

Syntax

```
#pragma argsused
```

Description

The **argsused** pragma is allowed only between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

1.1.1.4.8.6 #pragma codeseg

Syntax

```
#pragma codeseg <seg_name> <"seg_class"> <group>
```

Description

The **codeseg** directive lets you name the segment, class, or group where functions are allocated. If the pragma is used without any of its options, the default code segment is used for function allocation.

1.1.1.4.8.7 #pragma checkoption

Syntax

```
#pragma checkoption optionstring
```

Description

The **checkoption** pragma checks that the options in 'optionstring' are actually set. If they are not, an error is emitted:

```
E2471: pragma checkoption failed: options are not as expected
```

Use the **checkoption** pragma with an option string indicating the command-line options to check for. For example:

```
#pragma checkoption -a4 -b
```

#pragma checkoption optionstring is similar to **#pragma option** optionstring, which attempts to set the options in 'optionstring'.

1.1.1.4.8.8 #pragma comment

Syntax

```
#pragma comment (comment type, "string")
```

Description

The **comment** directive lets you write a comment record into an output file. The comment type can be one of the following values:

Value	Explanation
exestr	The linker writes string into an .obj file. Your specified string is placed in the executable file. Such a string is never loaded into memory but can be found in the executable file by use of a suitable file search utility.
lib	Writes a comment record into an .obj file. A library module that is not specified in the linker's response-file can be specified by the **comment** LIB directive. The linker includes the library module name specified in string as the last library. Multiple modules can be named and linked in the order in which they are named.
user	The compiler writes string into the .obj file. The linker ignores the specified string.

1.1.1.4.8.9 #pragma exit and #pragma startup

Syntax

```
#pragma startup function-name <priority>
#pragma exit function-name <priority>
```

Description

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the main function is called), or program exit (just before the program terminates through **_exit**).

The specified function-name must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as:

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

The optional priority parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100.

Warning: Do not use priority values less than 64. Priorities from 0 to 63 are reserved for ISO startup and shutdown mechanisms.

1.1.1.4.8.10 #pragma hdrfile

Syntax

```
#pragma hdrfile "filename.csm"
```

Description

This directive sets the name of the file in which to store precompiled headers.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option -H=filename or Use Precompiled Headers to change the name of the file used to store precompiled headers.

1.1.1.4.8.11 #pragma curious_george

Syntax

```
#pragma curious_george
```

Description

The **#pragma curious_george** directive prints out the ‘warning’: Please, No eating puzzle pieces!

This message will only be displayed in the IDE if the Compiler options for "Show general messages" is checked. The message can also be enabled or disabled using the compiler's **-wmsg** switch.

1.1.1.4.8.12 #pragma hdrstop

Syntax

```
#pragma hdrstop
```

Description

This directive terminates the list of header files eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers.

Precompiled header files can be shared between the source files of your project only if the **#include** directives before **#pragma hdrstop** are identical. Therefore, you get the best compiler performance if you include common header files of your project before the **#pragma hdrstop**, and specific ones after it. Make sure the **#include** directives before the **#pragma hdrstop** are identical in all the source files, or that there are only very few variations.

The integrated development environment generates code to enhance precompiled header performance. For example, after a New Application, source file "Unit1.cpp" will look like this (comments added):

```
#include <clxvcl.h>       // common header file
#pragma hdrstop      // terminate list here
```

Use this pragma directive only in source files. The pragma has no effect when it is used in a header file.

1.1.1.4.8.13 #pragma defineonoption And undefineonoption

Syntax

```
#pragma defineonoption name switch(es)CommandLineOptionsswitch(es)CommandLineOptions
#pragma undefineonoption name switch(es)CommandLineOptions
```

Description

These two pragmas allow you to alias one or more command-line options to a name. You can use most of the command-line switches in the pragma except those that take input such as –I and –L.

The **#pragma defineonoption** defines name if all specified switch(es) are on at the time of the pragma. Similarly, the **#pragma undefineonoption** undefines name if all the switch(es) are on at the time of the pragma.

For example, myfile.c includes the following code:

```
#pragma defineonoption OPTIMIZING -O2
#ifdef    OPTIMIZING
```

When you compile with bcc32 -c myfile.c, the output is:

```
There are no optimizations turned on
```

Now compile with bcc32 -c -O2 myfile.c, and the output is:

```
We are optimizing with -O2
```

You can use more than one switch. For instance, consider the following myfile.c, which demonstrates that you can undefine options:

```
#define SWITCHES_ARE_OFF
#pragma undefineonoption SWITCHES_ARE_OFF -O2 -c -P -C
```

Compile with bcc32 -c myfile.c, and the result is:

```
At least one of the switches is not turned on
```

Compile with bcc32 -c -P -O2 -C, and you'll see:

```
All switches are turned on
```

Or compile with bcc32 -P -c -C -O2 -v -f -H, and you'll see:

```
All switches are turned on
```

As you can see, the order of the switches is not important.

What not to do

You can only define or undefine one name, so the following syntax generates an error:

```
#pragma defineonoption DEF1 DEF2 -O2
```

You must use one name to define or undefine, and you must use at least one switch, so the following examples all generate errors:

```
#pragma defineonoption -O2
#pragma undefineonoption DEF1
```

1.1.1.4.8.14 #pragma inline

Syntax

#pragma inline

Description

This directive is equivalent to the -B command-line compiler option.

This is best placed at the top of the file, because the compiler restarts itself with the -B option when it encounters **#pragma inline**.

1.1.1.4.8.15 #pragma intrinsic

Syntax

```
#pragma intrinsic [-]function-name
```

Description

Use **#pragma intrinsic** to override command-line switches or project options to control the inlining of functions.

When inlining an intrinsic function, always include a prototype for that function before using it.

1.1.1.4.8.16 #pragma link

Syntax

```
#pragma link "[path]modulename[.ext]"
```

Description

The directive instructs the linker to link the file into an executable file.

By default, the linker searches for modulename in the local directory and any path specified by the **-L** option. You can use the path argument to specify a directory.

By default, the linker assumes a .obj extension.

1.1.1.4.8.17 #pragma message

Syntax

```
#pragma message ("text" ["text"["text" ...]])
#pragma message text
```

Description

Use **#pragma message** to specify a user-defined message within your program code.

The first form requires that the text consist of one or more string constants, and the message must be enclosed in parentheses. (This form is compatible with Microsoft C.) This form will output the constant contained between the double quotes regardless of whether it is a macro or not.

The second form uses the text following the **#pragma** for the text of the warning message. With this form of the **#pragma**, any macro references are expanded before the message is displayed.

The third form will output the macro-expanded value of text following the **#pragma**, if it is **#defined**. If is not **#defined**, you'll get an ill-formed pragma warning.

User-defined messages are displayed as messages, not warnings.

Display of user-defined messages is on by default and can be turned on or off with the `Show Messages` option. This option corresponds to the compiler's **-wmsg** switch.

Messages are only displayed in the IDE if Show general messages is checked on the C++ Project Properties under **Project**▸**Options**▸**Project Propertes**.

See Also

Preprocessor Directives (see page 148)

1.1.1.4.8.18 #pragma nopushoptwarn

Syntax

```
#pragma nopushoptwarn
```

Description

The **nopushoptwarn** pragma allows the use of pragma option push without pop and pragma option pop without push // in a file. Normally a warning would be given.

1.1.1.4.8.19 #pragma pack

Syntax

```
#pragma pack([{push | pop}[,]] [identifier[,]] [n])
```

Description

The **#pragma pack(n)**.directive is the same as using the #pragma option specifically with the -a compiler option. **n** is the byte

alignment that determines how the compiler aligns data in stored memory. For more information see the -a compiler option. **#pragma pack** can also be used with **push** and **pop** arguments, which provide the same functionality as the **#pragma option** directive using **push** and **pop**. The following table compares the use of **#pragma pack** with **#pragma option**.

#pragma pack	#pragma option
#pragma pack(n)	#pragma option -an
#pragma pack(push, n)	#pragma option push -an
#pragma pack(pop)	#pragma option pop

The **#pragma pack** directive also supports an identifier argument which must be used in combination with either push or pop.

#pragma pack with no arguments

```
#pragma pack()
```

Using **#pragma pack** with no arguments will set the packing size to the starting –aX alignment (which defaults to 8). The starting -aX alignment is considered the alignment at the start of the compile AFTER all command-line options have been processed.

#pragma pack using a value for n

```
#pragma pack(8)
```

Using **#pragma pack** with a value for 'n' will set the current alignment to 'n'. Valid alignments for ‘n’ are: 1,2,4,8, and 16.

#pragma pack using push

```
#pragma pack(push)
```

Using **#pragma pack** with push will push the current alignment on an internal stack of alignments.

#pragma pack using push, identifier

```
#pragma pack(push, ident)
```

Using **#pragma pack** with push and an identifier will associate the pushed alignment with 'identifier'.

#pragma pack using push and n

```
#pragma pack(push, 8)
#pragma pack(push, ident, 8)
```

Using **#pragma pack** with push with a value for 'n', will execute pragma pack push or pragma pack push identifier, and afterwards set the current alignment to 'n'.

#pragma pack using pop

```
#pragma pack(pop)
```

Using **#pragma pack** with pop will pop the alignment stack and set the alignment to the last alignment pushed. If the pop does not find a corresponding push, the entire stack of alignments is popped, and a warning is issued, and the alignment reverts to the starting -aX alignment..

#pragma pack using pop, identifier

```
#pragma pop(pop, ident)
```

Using **#pragma pack** with pop and an identifier will pop the stack until the identifier is found and set the alignment to the alignment pushed by the previous corresponding **#pragma pack(push, identifier)**. If the pop with identifier does not find the corresponding push with an identifier, the entire stack of alignments is popped, and a warning is issued to the user:

```
W8083: Pragma pack pop with no matching pack push
```

The alignment will then be reset to the starting -aX alignment.

#pragma pack using pop and n

```
#pragma pack(pop, 8)
#pragma pack(pop, ident, 8)
```

Using **#pragma pack** with pop and a value for 'n', will execute pragma pack pop or pragma pack pop identifier. Afterwards the current alignment is set to 'n', unless pop fails to find a corresponding push, in which case 'n' is ignored, a warning is issued, and the alignment reverts to the starting -aX alignment.

Error conditions

Specifying an 'identifier' without push or pop is an error.

Specifying an alignment different from 1,2,4,8,16 is an error.

Warning conditions

Using **#pragma pop** without a corresponding push issues a warning.

1.1.1.4.8.20 #pragma package

Syntax

```
#pragma package(smart_init)
#pragma package(smart_init, weak)
```

Description: smart_init argument

The **#pragma package(smart_init)** assures that packaged units are initialized in the order determined by their dependencies. (Included by default in package source file.) Typically, you would use the **#pragma package** for .cpp files that are built as packages.

This pragma affects the order of initialization of that compilation unit. For units, initialization occurs in the following order:

- 1. By their "uses" dependencies, that is, if unitA depends on unitB, unitB must be initialized before unitA.
- 2. The link order.
- 3. Priority order within the unit.

For regular object files (those not built as units), initialization happens first according to priority order and then link order. Changing the link order of the object files changes the order in which the global object constructors get called.

The following examples show how the initialization differs between units and regular object files.

Take as an example three unit files, A, B and C that are "smart initialized" with **#pragma package(smart_init)** and have priority values (defined by the priority parameter of the #pragma startup) set of 10, 20 and 30. The functions are named according to their priority value and the parent object file, so the names are a10, a20, a30, b10, and so on.

Since all three are units, and if A uses B and C and the link order is A, B then C, the order of initialization is:

B10 B20 B30 C10 C20 C30 A10 A20 A30

If the above were object files, not units, the order would be:

A10 B10 C10 A20 B20 C20 A30 B30 C30

The .cpp files that use **#pragma package(smart_init)** also require that any **#pragma link** references to other object files from a .cpp file that declares **#pragma package(smart_init)**, must be resolved by a unit. **#pragma link** references to non object files can still be resolved by libraries, etc.

Description: weak packages

The **#pragma package(smart_init, weak)** directive affects the way an object file is stored in a package's .bpi and .bpl files. If **#pragma package(smart_init, weak)** appears in a unit file, the compiler omits the unit from BPLs when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is

said to be "weakly packaged".

#pragma package(smart_init, weak) is used to eliminate conflicts among packages that may depend on the same external library.

Unit files containing the **#pragma package(smart_init, weak)** directive must not have global variables.

For more information about using weak packages, see Weak packaging.

See Also

Preprocessor Directives (see page 148)

1.1.1.4.8.21 #pragma obsolete

Syntax

```
#pragma obsolete identifier
```

Description

Use **#pragma obsolete** to give a warning the first time the preprocessor encounters identifier in your program code after the pragma. The warning states that the identifier is obsolete.

1.1.1.4.8.22 #pragma option

Syntax

```
#pragma option options
#pragma option push options
#pragma option pop
#pragma nopushoptwarn
```

Description

Use **#pragma option** to include command-line options within your program code. **#pragma option** can also be used with the push or pop arguments.

#pragma option [options...]

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. For example, both of the following are valid:

```
#pragma option -C
#pragma option -C -A
```

Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, and then return it to its default, without having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include:

B	c	dname
Dname=string	efilename	E
Fx	h	Ifilename
lexset	M	o
P	Q	S
T	Uname	V

X	Y	

- You can use **#pragma**s, **#include**s, **#define**, and some **#if**s in the following cases:
- Before the use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an **#if**, **#ifdef**, **#ifndef** or **#elif** directive.
- Before the occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can appear only in a #**pragma option** command before these events. These options are:

Efilename	f	i#
m*	npath	ofilename
U	W	z

*

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

1	h	r
2	k	rd
A	N	v
Ff	O	y
G	p	Z

The following options can be changed at any time and take effect immediately:

A	gn	zE
b	jn	zF
C	K	zH
d	wxxx	

The options can appear followed by a dot (.) to reset the option to its command-line state.

#pragma option using push or pop

The **#pragma option** directive can also be used with the push and pop arguments to enable you to easily modify compiler directives.

Using the **#pragma option push** allows you to save all (or a selected subset of) options before including any files that could potentially change many compiler options and warnings, and then, with the single statement, **#pragma option pop**, return to the previous state. For example:

```
#pragma option push
#include <theworld.h>
#pragma option pop
#include "mystuff.h"
```

The **#pragma option push** directive first pushes all compiler options and warning settings on a stack and then handles any options (if supplied). The following examples show how the **#pragma option push** can be used with or without options:

```
#pragma option push -C -A
#pragma option push
```

The **#pragma option pop** directive changes compiler options and warnings by popping the last set of options and warnings from

the stack. It gives a warning, "Pragma option pop with no matching option push", if the stack is empty, in which case nothing happens.

The following generates a warning about an empty stack:

```
#pragma option push
#pragma option pop
#pragma option pop      /* Warning */
```

We do not recommend it, but you can turn off this warning with the directive: #pragma warn -nop.

If you try to specify any options after pop, you get the error, "Nothing allowed after pragma option pop." For example, the following produces an error:

```
#pragma option pop -C/* ERROR */
```

If your stack of pushed options is not the same at the start of a file as at the end of a file, you receive a warning: "Previous options and warnings not restored." To turn off this warning, use the directive, **#pragma nopushoptwarn** .

1.1.1.4.8.23 #pragma resource

Syntax

```
#pragma resource "*.xdfm"
```

Description

This pragma causes the file to be marked as a form unit and requires matching .dfm and header files. All such files are managed by the IDE.

If your form requires any variables, they must be declared immediately after the pragma resource is used. The declarations must be of the form

TFormName *Formname;

1.1.1.4.8.24 #pragma warn

Syntax

```
#pragma warn [+|-|.]xxx
```

Description

The **warn** pragma lets you override specific **-wxxx** command-line options or check Display Warnings. **xxx** is the three-letter or four-digit message identifier used by the command-line option.

1.1.1.4.9 Predefined Macros Overview

This section contains Predefined Macros Overview topics.

1.1.1.4.9.1 Predefined Macros

The compiler predefines certain global identifiers, known as manifest constants. Most global identifiers begin and end with two underscores (__).

Note: For readability, underscores are often separated by a single blank space. In your source code, you should never insert whitespace between underscores.

Macro	Value	Description
__BCOPT__	1	Defined in any compiler that has an optimizer.
__BCPLUSPLUS__	0x0570	Defined if you've selected C++ compilation; will increase in later releases.
__BORLANDC__	0x0570	Version number.
__CDECL__	1	Defined if Calling Convention is set to cdecl; otherwise undefined.
_CHAR_UNSIGNED	1	Defined by default indicating that the default **char** is **unsigned char**. Use the -K compiler option to undefine this macro.
__CODEGUARD__		Defined whenever one of the CodeGuard compiler options is used; otherwise it is undefined.
__CONSOLE__	1	When defined, the macro indicates that the program is a console application.
_CPPUNWIND	1	Enable stack unwinding. This is true by default; use **-xd-**!ALink(OSCGExceptions1) to disable.
__cplusplus	1	Defined if in C++ mode; otherwise, undefined.
__DATE__	String literal	Date when processing began on the current file.
__DLL__	1	Defined whenever the -WD compiler option is used; otherwise it is undefined.
__FILE__	String literal	Name of the current file being processed.
__FLAT__	1	Defined when compiling in 32-bit flat memory model.
__FUNC__	String literal	Name of the current function being processed. More details.
__LINE__	Decimal constant	Number of the current source file line being processed.
_M_IX86	0x12c	Always defined. The default value is 300. You can change the value to 400 or 500 by using the /4 or /5 compiler options.
__MT__	1	Defined only if the -tWM option is used. It specifies that the multithread library is to be linked.
__PASCAL__	1	Defined if Calling Convention is set to Pascal; otherwise undefined.
__STDC__	1	Defined if you compile with the -A compiler option; otherwise, it is undefined.
__TCPLUSPLUS__	0x0570	Version number.
__TEMPLATES__	1	Defined as 1 for C++ files (meaning that templates are supported); otherwise, it is undefined.
__TIME__	String literal	Time when processing began on the current file.
__TLS__	1	Thread Local Storage. Always true.
__TURBOC__	0x0570	Will increase in later releases.
_WCHAR_T		Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.

_WCHAR_T_DEFINED		Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.
_Windows		Defined when compiling on the Windows platform.
__WIN32__	1	Defined for console and GUI applications on the Windows platform.
__linux__		Defined when compiling on the Linux platform.

Note: **__DATE__**, **__FILE__** , **__FUNC__**, **__LINE__**, **__STDC__**, and **__TIME__** cannot be redefined or undefined.

See Also

Side Effects And Other Dangers (see page 152)

Using The Backslash (\) For Line Continuation (see page 152)

Converting To Strings With # (see page 151)

1.1.1.5 Keywords, By Category

This section contains Keywords, By Category topics.

1.1.1.5.1 C++ Specific Keywords

This section contains C++ Specific Keyword topics.

1.1.1.5.1.1 asm, _asm, __asm

Category

Keyword extensions, C++-Specific Keywords

Syntax

```
asm <opcode> <operands> <; or newline>_asm__asm
_asm <opcode> <operands> <; or newline>
__asm <opcode> <operands> <; or newline>
```

Description

Use the **asm,** **_asm**, or **__asm** keyword to place assembly language statements in the middle of your C or C++ source code. Any C++ symbols are replaced by the appropriate assembly language equivalents.

You can group assembly language statements by beginning the block of statements with the **asm** keyword, then surrounding the statements with braces ({}).

1.1.1.5.1.2 bool, false, true

Category

C++-Specific Keywords

Syntax

```
bool <identifier>;
```

Description

Use **bool** and the literals **false** and **true** to perform Boolean logic tests.

The **bool** keyword represents a type that can take only the value **false** or **true**. The keywords **false** and **true** are Boolean literals with predefined values. **false** is 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**.

1.1.1.5.1.3 catch

Category

Statements, C++-Specific Keywords

Syntax

```
catch (exception-declaration) compound-statement
```

Description

The exception handler is indicated by the **catch** keyword. The handler must be used immediately after the statements marked by the try keyword. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

1.1.1.5.1.4 class

Category

C++-Specific Keywords, Type specifiers

Syntax

```
<classkey> <classname> <baselist> { <member list> }
```

- <classkey> is either a class**,** struct, or union.
- <classname> can be any name unique within its scope.
- <baselist> lists the base class(es) that this class derives from. <baselist> is optional
- <member list> declares the class's data members and member functions.

Description

Use the **class** keyword to define a C++ class.

Within a class:

- the data are called data members
- the functions are called member functions

See Also

Private (see page 173)

Protected (see page 173)

Public (see page 174)

1.1.1.5.1.5 explicit

Category

C++-Specific Keywords

Syntax

```
explicit <single-parameter constructor declaration>
```

Description

Normally, a class with a single-parameter constructor can be assigned a value that matches the constructor type. This value is automatically (implicitly) converted into an object of the class type to which it is being assigned. You can prevent this kind of implicit conversion from occurring by declaring the constructor of the class with the **explicit** keyword. Then all objects of that class must be assigned values that are of the class type; all other assignments result in a compiler error.

Objects of the following class can be assigned values that match the constructor type or the class type:

```
class X {
public:
X(int);
X(const char*, int = 0);
};
```

Then, the following assignment statements are legal.

```
void f(X arg) {
X a = 1;
X B = "Jessie";
a = 2;
}
```

However, objects of the following class can be assigned values that match the class type only:

```
class X {
public:
explicit X(int);
explicit X(const char*, int = 0);
};
```

The **explicit** constructors then require the values in the following assignment statements to be converted to the class type to which they are being assigned.

```
void f(X arg) {
X a = X(1);
X b = X("Jessie",0);
a = X(2);
}
```

1.1.1.5.1.6 friend

Category

C++-Specific Keywords

Syntax

```
friend <identifier>;
```

Description

Use **friend** to declare a function or class with full access rights to the private and protected members of the class, without being a member of that class. The outside class has fill access to the class that declares that outside class a friend.

In all other respects, the **friend** is a normal function in terms of scope, declarations, and definitions.

1.1.1.5.1.7 inline

Category

C++-Specific Keywords

Syntax

```
inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the **inline** keyword to declare or define C++ inline functions.

Inline functions are best reserved for small, frequently used functions.

1.1.1.5.1.8 mutable

Category

C++-Specific Keywords, Storage class specifiers

Syntax

```
mutable <variable name>;
```

Description

Use the **mutable** specifier to make a variable modifiable even though it is in a **const**-qualified expression.

Using the mutable Keyword

Only class data members can be declared mutable. The **mutable** keyword cannot be used on **static** or **const** names. The purpose of **mutable** is to specify which data members can be modified by **const** member functions. Normally, a **const** member function cannot modify data members.

See Also

Const (see page 46)

1.1.1.5.1.9 namespace

Category

C++-Specific Keywords

Description

Most real-world applications consist of more than one source file. The files can be authored and maintained by more than one developer. Eventually, the separate files are organized and linked to produce the final application. Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as function or class body, or translation unit) must share the same global namespace. Therefore, multiple definitions of names discovered while linking separate modules require some way to distinguish each name. The solution to the problem of name clashes in the global scope is provided by the C++ namespace mechanism.

The namespace mechanism allows an application to be partitioned into a number of subsystems. Each subsystem can define and operate within its own scope. Each developer is free to introduce whatever identifiers are convenient within a subsystem without worrying about whether such identifiers are being used by someone else. The subsystem scope is known throughout the application by a unique identifier.

It only takes two steps to use C++ namespaces. The first is to uniquely identify a namespace with the keyword **namespace**. The

second is to access the elements of an identified namespace by applying the **using** keyword.

1.1.1.5.1.10 operator

Category

Operators, C++-Specific Keywords

Syntax

```
operator <operator symbol>( <parameters> )
{
<statements>;
}
```

Description

Use the **operator** keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as *this is implicitly the first argument.

When you overload an operator as a friend, you can specify two arguments.

See Also

Class (see page 170)

1.1.1.5.1.11 private

Category

C++-Specific Keywords

Syntax

```
private: <declarations>
```

Description

Access to **private** class members is restricted to member functions within the class, and to **friend** classes.

Class members are **private** by default.

Structure (**struct**) and **union** members are **public** by default. You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

See Also

Class (see page 170)

Friend (see page 171)

Protected (see page 173)

1.1.1.5.1.12 protected

Category

C++-Specific Keywords

Syntax

```
protected: <declarations>
```

Description

Access to **protected** class members is restricted to member functions within the class, member functions of derived classes, and to **friend** classes.

Structure (**struct**) and **union** members are **public** by default. You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

See Also

Class (see page 170)

Friend (see page 171)

Private (see page 173)

1.1.1.5.1.13 public

Category

C++-Specific Keywords

Syntax

```
public: <declarations>
```

Description

A **public** class member can be accessed by any function.

Members of a struct or union are **public** by default.

You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

See Also

Class (see page 170)

Friend (see page 171)

Private (see page 173)

Protected (see page 173)

__published (see page 188)

Struct (see page 199)

1.1.1.5.1.14 this

Category

C++-Specific Keywords

Example

```
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 ->.

See Also

Class (see page 170)

1.1.1.5.1.15 throw

Category

Statements, C++-Specific Keywords

Syntax

```
throw assignment-expression
```

Description

When an exception occurs, the throw expression initializes a temporary object of the type T (to match the type of argument arg) used in throw(T arg). Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

See Also

Catch (see page 170)

1.1.1.5.1.16 try

Category

Statements, C++-Specific Keywords

Syntax

```
try compound-statement handler-list
```

Description

The **try** keyword is supported only in C++ programs. Use __try in C programs. C++ also allows __**try**.

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is 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.

See Also

Catch (see page 170)

1.1.1.5.1.17 typeid

Category

Operators, C++-Specific Keywords

Syntax

```
typeid( expression )
typeid( type-name )
```

Description

You can use **typeid** to get runtime identification of types and expressions. A call to **typeid** returns a reference to an object of type **const** type_info. The returned object represents the type of the **typeid** operand.

If the **typeid** operand is a dereferenced pointer or a reference to a polymorphic type, **typeid** returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, **typeid** returns an object that represents the static type.

You can use the **typeid** operator with fundamental data types as well as user-defined types.

When the **typeid** operand is a Delphi class object/reference, **typeid** returns the static rather than runtime type.

If the **typeid** operand is a dereferenced NULL pointer, the Bad_typeid exception is thrown.

1.1.1.5.1.18 typename

Category

C++-Specific Keywords

Syntax 1

```
typename identifier
```

Syntax 2

```
template < typename identifier  > class identifier
```

Description

Use the syntax 1 to reference a type that you have not yet defined. See example 1.

Use syntax 2 in place of the **class** keyword in a template declaration. See example 2.

Note: When using the **typename** keyword with templates, the compiler will not always report an error in cases where the ANSI standard requires the typename keyword. The compiler will flag the omission of typename when the compiler is invoked with the **-A** switch. For example, given the following code:

```
#include <stdio.h>
struct A{ typedef int AInt; };
```

Note: The compiler will flag the omission of typename when the compiler is invoked with the **-A** switch.

Note: Compile with: bcc32 (no -A switch)

```
bc++bcc32 test.cpp
```

The result is no error. The Compiler should not assume AInt is a **typename**, but it does unless **-A** switch is used

Note: Compile with: bcc32 (**-A** switch)

```
bc++bcc32 -A test.cpp
```

The result is:

Error E2089 47071.cpp 7: Identifier 'AInt' cannot have a type qualifier

Error E2303 47071.cpp 7: Type name expected

Error E2139 47071.cpp 7: Declaration missing ;

Both results are as expected.

1.1.1.5.1.19 using (declaration)

Category

C++-Specific Keywords

Description

You can access namespace members individually with the using-declaration syntax. When you make a using declaration, you add the declared identifier to the local namespace. The grammar is

using-declaration:

using :: unqualified-identifier;

1.1.1.5.1.20 virtual

Category

C++-Specific Keywords

Syntax

```
virtual class-namevirtual
virtual function-name
```

Description

Use the **virtual** keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as **virtual**, you can redefine it in any derived class, even if the number and type of arguments are the same.

The redefined function overrides the base class function.

1.1.1.5.1.21 wchar_t

Category

C++-Specific Keywords, Type specifiers

Syntax

```
wchar_t <identifier>;
```

Description

In C++ programs, **wchar_t** is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In CodeGear C++, A **wchar_t** type is the same size, signedness, and alignment requirement as an unsigned short type.

1.1.1.5.2 C++ Builder Keyword Extensions

This section contains C++ Builder Keyword Extension topics.

1.1.1.5.2.1 __automated

Category

Keyword extensions

Syntax

```
_automated: <declarations>
```

Description

The visibility rules for automated members are identical to those of public members. The only difference between automated and public members is that OLE automation information is generated for member functions and properties that are declared in an automated section. This OLE automation type information makes it possible to create OLE Automation servers.

- For a member function, the types of all member function parameters and the function result (if any) must be automatable. Likewise, for a property, the property type and the types of any array property parameters must be automatable. The automatable types are: Currency, OleVariant, DelphiInterface, double, int, float, short, String, TDateTime, Variant, and unsigned short. Declaring member functions or properties that use non-automatable types in an __automated section results in a compile-time error.
- Member function declarations must use the __fastcall calling convention.
- Member functions can be virtual.
- Member functions may add __dispid(constant int expression) after the closing parenthesis of the parameter list.
- Property declarations can only include access specifiers (__dispid, read, and write). No other specifiers (index, stored, default, nodefault) are allowed.
- Property access specifiers must list a member function identifier. Data member identifiers are not allowed.
- Property access member functions must use the __fastcall calling convention.
- Property overrides (property declarations that don't include the property type) are not allowed.

See Also

__dispid (see page 189)

__closure (see page 179)

__property (see page 188)

__published (see page 188)

1.1.1.5.2.2 __classid

Category

Operators, Keyword extensions

Syntax

```
__classid(classType)
```

Description

The __**classid** operator was added to support the VCL framework. Normally, programmers should not directly use this operator. For more information, see the keyword extensions.

See Also

__property (see page 188)

1.1.1.5.2.3 __closure

Category

Keyword extensions

Syntax

```
<type> ( __closure * <id> ) (<param list>);
```

Description

The keyword **__closure** was added to support the VCL framework and is used when declaring event handler functions. For more information, see the keyword extensions.

See Also

__property (see page 188)

1.1.1.5.2.4 __declspec

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec(<decl-modifier>)
```

Description

For a list of **__declspec** keyword arguments used for the VCL framework, see VCL class declarations.

Use the **__declspec** keyword to indicate the storage class attributes for a variable or function.

The **__declspec** keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The **__declspec** keyword and its argument can appear anywhere in the declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);                      // illegal
void __export f(void)                       // correct
void __declspec(dllexport) f(void);         // correct
__declspec(dllexport)void f(void);          // correct
class __declspec(dllexport) ClassName { }   // correct
```

In addition to the arguments listed above, the supported decl-modifier arguments are:

- dllexport
- dllimport
- naked
- noreturn
- nothrow
- novtable
- property
- selectany

- thread
- uuid

These arguments are equivalent to the following storage class attribute keywords.

Argument	Comparable keyword
dllexport	__export
dllimport	__import
thread	__thread

See Also

_export (see page 184)

_import (see page 187)

__declspec(dllexport) (see page 180)

__declspec(dllimport) (see page 180)

__declspec(naked) (see page 181)

__declspec(noreturn) (see page 181)

__declspec(nothrow) (see page 182)

__declspec(novtablet) (see page 182)

__declspec(property) (see page 182)

__declspec(selectany) (see page 183)

__declspec(thread) (see page 184)

__declspec(uuid) (see page 184)

1.1.1.5.2.5 __declspec(dllexport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllexport ) declarator
```

The **dllexport** storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to export functions, data, and objects from a DLL. This attribute explicitly defines the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as **dllexport** eliminates the need for a module-definition (.DEF) file, at least with respect to the specification of exported functions.

Note: **dllexport** replaces the **__export** keyword.

1.1.1.5.2.6 __declspec(dllimport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllimport ) declarator
```

The **dllimport** storage-class attribute is used for Microsoft C and C++ language compatability. This attribute enables you to import functions, data, and objects to a DLL

Note: **Note**: **dllimport** replaces the __**import** keyword.

1.1.1.5.2.7 __declspec(naked)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( naked ) declarator
```

Use of the naked argument suppresses the prolog/epilog code. Be aware when using __**declspec(naked)** that it does not set up a normal stack frame. A function with __**declspec(naked)** will not preserve the register values that are normally preserved. It is the programmer's responsibility to conform to whatever conventions the caller of the function expects.

You can use this feature to write your own prolog/epilog code using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

The **naked** attribute is relevant only to the definition of a function and is not a type modifier.

Example

This code defines a function with the **naked** attribute:

```
// Example of the naked attribute
__declspec( naked ) int func( formal_parameters )
{
// Function body
}
```

1.1.1.5.2.8 __declspec(noreturn)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( noreturn ) declarator
```

This __**declspec** attribute tells the compiler that a function does not return. As a consequence, the compiler knows that the code following a call to a __**declspec(noreturn)** function is unreachable.

If the compiler finds a function with a control path that does not return a value, it generates a warning. If the control path cannot be reached due to a function that never returns, you can use __**declspec(noreturn)** to prevent this warning or error.

Example

Consider the following code. The **else** clause does not contain a return statement, so the programmer declares fatal as __**declspec(noreturn)** to avoid an error or warning message.

```
__declspec(noreturn) extern void fatal ()
{
   // Code omitted
}
int foo()
{
   if(...)
```

```
      return 1;
   else if(...)
      return 0;
   else
      fatal();
}
```

1.1.1.5.2.9 __declspec(nothrow)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( nothrow ) declarator
```

This is a **__declspec** extended attribute that can be used in the declaration of functions. This attribute tells the compiler that the declared function and the functions it calls never throw an exception. With the synchronous exception handling model, now the default, the compiler can eliminate the mechanics of tracking the lifetime of certain unwindable objects in such a function, and significantly reduce the code size.

The following three declarations are equivalent:

```
#define WINAPI __declspec(nothrow) __stdcall
void WINAPI foo1();
void __declspec(nothrow) __stdcall foo2();
void __stdcall foo3() throw();
```

Using **void __declspec(nothrow) __stdcall foo2();** has the advantage that you can use an API definition, such as the illustrated by the **#define** statement, to easily specify nothrow on a set of functions. The third declaration, **void __stdcall foo3() throw();** is the syntax defined by the C++ standard.

1.1.1.5.2.10 __declspec(novtable)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( novtable ) declarator
```

This form of **_declspec** can be applied to any class declaration, but should only be applied to pure interface classes, that is classes that will never be instantiated on their own. The **_declspec** stops the compiler from generating code to initialize the vfptr in the constructor(s) and destructor of the class. In many cases, this removes the only references to the vtable that are associated with the class and, thus, the linker will remove it. Using this form of **_declspec** can result in a significant reduction in code size

1.1.1.5.2.11 __declspec(property)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

This attribute can be applied to non-static “virtual data members” in a class or structure definition. The compiler treats these “virtual data members” as data members by changing their references into function calls.

When the compiler sees a data member declared with this attribute on the right of a member-selection operator ("**.**" or "**->**"), it converts the operation to a **get** or **put** function, depending on whether such an expression is an l-value or an r-value. In more complicated contexts, such as "**+=**", a rewrite is performed by doing both **get** and **put**.

This attribute can also be used in the declaration of an empty array in a class or structure definition.

Example

__**declspec**(property(get=GetX, put=PutX)) int x[];

The above statement indicates that x[] can be used with one or more array indices. In this case:

```
i=p->x[a][b]
```

will be turned into:

```
i=p->GetX(a, b),
```

and

```
p->x[a][b] = i
```

will be turned into

```
p->PutX(a, b, i);
```

1.1.1.5.2.12 __declspec(selectany)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( selectany ) declarator
```

A global data item can normally be initialized only once in an application or library. This attribute can be used in initializing global data defined by headers, when the same header appears in more than one source file.

Note This attribute can only be applied to the actual initialization of global data items that are externally visible.

Example

This code shows how to use the **selectany** attribute:

//Correct - x1 is initialized and externally visible

__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so

//x2 is not visible externally (This is OK in C, since

//const is not by default static in C)

const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible

extern const __**declspec**(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible

extern const int x4;

const __**declspec**(selectany) int x4=4;

//Incorrect - __**declspec**(selectany) is applied to the uninitialized //declaration of x5 extern __**declspec**(selectany) int x5;

1.1.1.5.2.13 __declspec(thread)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( thread ) declarator
```

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process allocates storage for thread-specific data.

The **thread** extended storage-class modifier is used to declare a thread local variable. The **thread** attribute must be used with the __**declspec** keyword.

1.1.1.5.2.14 __declspec(uuid(“ComObjectGUID”))

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( uuid(“ComObjectGUID”) ) declarator
```

The compiler attaches a GUID to a class or structure declared or defined (full COM object definitions only) with the **uuid** attribute. The **uuid** attribute takes a string as its argument. This string names a GUID in normal registry format with or without the **{ }** delimiters. For example:

struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;

struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;

This attribute can be applied in a redeclaration. This allows the system headers to supply the definitions of interfaces such as **IUnknown**, and the redeclaration in some other header (such as COMDEF.H) to supply the GUID.

The keyword __uuidof can be applied to retrieve the constant GUID attached to a user-defined type.

1.1.1.5.2.15 __except

Category

Statements, Keyword extensions

Syntax

```
__except (expression) compound-statement
```

Description

The __except keyword specifies the action that should be taken when the exception specified by expression has been raised.

See Also

__finally (see page 185)

__try (see page 189)

1.1.1.5.2.16 _export, __export

Category

Modifiers, Keyword extensions

Form 1

```
class _export <class name>
```

Form 2

```
return_type _export <function name>
```

Form 3

```
data_type _export <data name>
```

Description

These modifiers are used to export classes, functions, and data.

The linker enters functions flagged with **_export** or **__export** into an export table for the module.

Using **_export** or **__export** eliminates the need for an EXPORTS section in your module definition file.

Functions that are not modified with **_export** or **__export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note: If you use **_export** or **__export** to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need a module definition file.

See Also

Compiler Options And The _export Keyword

Exportable Functions

1.1.1.5.2.17 __finally

Category

Statements, Keyword extensions

Syntax

```
__finally {compound-statement}
```

Description

The **__finally** keyword specifies actions that should be taken regardless of how the flow within the preceding __try exits.

The following is the code fragment shows how to use the **try/__finally** construct:

```
#include <stdio.h>#include <string.h>#include <windows.h>class
Exception{public:Exception(char* s = "Unknown"){what = strdup(s);      }Exception(const
Exception& e ){what = strdup(e.what); } ~Exception()                    {free(what);         }
char* msg() const             {return what;           }private:char* what;};int main(){float
e, f, g;try  {try    {f = 1.0;g = 0.0;try      {puts("Another exception:");e = f / g;
}__except(EXCEPTION_EXECUTE_HANDLER)      {puts("Caught a C-based
exception.");throw(Exception("Hardware error: Divide by 0"));      }    }catch(const
Exception& e)    {printf("Caught C++ Exception: %s :\n", e.msg());    }  }__finally
{puts("C++ allows __finally too!");  }return e;}
#include <string.h>
#include <windows.h>
class Exception
{
public:
Exception(char* s = "Unknown"){what = strdup(s);      }
```

```
Exception(const Exception& e ){what = strdup(e.what); }
 ~Exception()                  {free(what);         }
  char* msg() const            {return what;          }
private:
char* what;
};
int main()
{
float e, f, g;
try
  {
try
    {
f = 1.0;
g = 0.0;
try
      {
puts("Another exception:");
e = f / g;
      }
__except(EXCEPTION_EXECUTE_HANDLER)
      {
puts("Caught a C-based exception.");
throw(Exception("Hardware error: Divide by 0"));
      }
    }
catch(const Exception& e)
    {
printf("Caught C++ Exception: %s :\n", e.msg());
    }
  }
__finally
  {
puts("C++ allows __finally too!");
  }
return e;
}
#include <iostream>#include <stdexcept>using namespace std;class MyException : public
exception {public:virtual const char *what() const throw() {return("MyException
occurred.");}};// Give me any integer...void myFunc(int a){ MyException e;   // ...but not
that one.   if(a == 0)     throw(e);}void main(){   int g;   // Note __finally must be in its
own try block (with no preceding catch).   try {     try {       g = 0;       myFunc(g);
}     catch(MyException &e) {       cout << "Exception: " << e.what() << endl;     }   }
__finally {       cout << "Finally block reached." << endl;   }
#include <stdexcept>
using namespace std;
class MyException : public exception {
public:
virtual const char *what() const throw() {
return("MyException occurred.");
}
};
// Give me any integer...
void myFunc(int a)
{
 MyException e;
   // ...but not that one.
   if(a == 0)
     throw(e);
}
void main()
{
   int g;
   // Note __finally must be in its own try block (with no preceding catch).
   try {
     try {
```

```
        g = 0;
        myFunc(g);
      }
      catch(MyException &e) {
        cout << "Exception: " << e.what() << endl;
      }
    }
    __finally {
        cout << "Finally block reached." << endl;
    }
}
```

Running the above program results in the following:

```
Another exception:Caught a C-based exception.Caught C++ exception[Hardware error: Divide by
0]C++ allows __finally too!Exception: MyException occurred.Finally block reached.
Caught a C-based exception.Caught C++ exception[Hardware error: Divide by 0]C++ allows
__finally too!Exception: MyException occurred.Finally block reached.
```

1.1.1.5.2.18 import, _import, __import

Category

Modifiers, Keyword extensions

Form 1

```
class _import <class name>class __import
class __import <class name>
```

Form 2

```
return_type _import <function name>__import
return_type __import <function name>
```

Form 3

```
data_type _import <data name>__import
data_type __import <data name>
```

Description

This keyword can be used as a class, function, or data modifier.

1.1.1.5.2.19 __inline

Category

Keyword extensions

Syntax

```
__inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the **__inline** keyword to declare or define C or C++ inline functions. The behavior of the **__inline** keyword is identical to that of the **inline** keyword, which is only supported in C++.

Inline functions are best reserved for small, frequently used functions.

1.1.1.5.2.20 __msfastcall

Category

Modifiers, Keyword extensions

Syntax

```
__msfastcall <function-name>
```

Description

This calling convention emulates the Microsoft implementation of the fastcall calling convertion. The first two DWORD or smaller arguments are passed in ECX and EDX registers, all other arguments are passed from right to left. The called function is responsible for removing these arguments from the stack.

1.1.1.5.2.21 __msreturn

Category

Modifiers, Keyword extensions

Syntax

```
__msreturn <function-name>
```

Description

This calling convention is used for Microsoft compatible __**fastcall** calling convention return values. Structures with a size that is greater than 4 bytes and less than 9 bytes, and having at least one of its members sized 4 bytes or larger, are returned in EAX/EDX.

1.1.1.5.2.22 __property

Category

Keyword extensions

Syntax

```
<property declaration> ::=
     __property <type> <id> [ <prop dim list> ] = "{" <prop attrib list> "}"

<prop dim list> ::=
     "[" <type> [ <id> ] "]" [ <prop dim list> ]

<prop attrib list> ::=
     <prop attrib> [ , <prop attrib list> ]

<prop attrib> ::=
     read = <data/function id> |
     write = <data/function id> |
     stored = <data/function id> |
     stored = <boolean constant> |
     default = <constant> |
     nodefault |
     index = <const int expression>
```

Description

The __**property** keyword was added to support the VCL

See Also

__closure (see page 179)

1.1.1.5.2.23 __published

Category

Keyword extensions

Syntax

```
__published: <declarations>
```

Description

The **__published** keyword was added to support the VCL.

See Also

__closure (see page 179)

__dispid (see page 189)

1.1.1.5.2.24 __try

Category

Statements, Keyword extensions

Syntax

```
__try compound-statement handler-list__try
__try compound-statement termination-statement
```

Description

The **__try** keyword is supported in both C and C++ programs. You can also use try in C++ programs.

A block of code in which an exception can occur must be prefixed by the keyword **__try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the normal program flow is interrupted. The program begins a search for a handler that matches the exception. If the exception is generated in a C module, it is possible to handle the structured exception in either a C module or a C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

See Also

Catch (see page 170)

__finally (see page 185)

1.1.1.5.3 Modifiers

This section contains Modifier topics.

1.1.1.5.3.1 __dispid

Category

Modifiers

Syntax

```
__dispid(constant int expression)
```

Description

A member function that has been declared in the __automated section of a class can include an optional __dispid(constant int expression) directive. The directive must be declared after the closing parenthesis of the parameter list.

The constant int expression gives the Automation dispatch ID of the member function or property. If a dispid directive is not used, the compiler automatically picks a number one larger than the largest dispatch ID used by any member function or property in the class and its base classes.

Specifying an already-used dispatch ID in a dispid directive causes a compile-time error.

See Also

__closure (see page 179)

__property (see page 188)

1.1.1.5.4 Operators

This section contains Operator topics.

1.1.1.5.4.1 if, else

Category

Operators

Syntax

```
if ( <condition1> )  <statement1>
if ( <condition1> )  <statement1>;
 else  <statement2>;
if ( <condition1> )  <statement1>;
 else if ( <condition2> )  <statement2>;
 else  <statement3>;
```

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.

1.1.1.5.5 Special Types

This section contains Special Type topics.

1.1.1.5.6 Statements

This section contains Statement topics.

1.1.1.5.6.1 break

Category

Statements

Syntax

```
break;
```

Description

Use the **break** statement within loops to pass control to the first statement following the innermost **switch**, **for**, **while**, or **do** block.

See Also

Continue (see page 192)

Do (see page 192)

For (see page 193)

Switch (see page 194)

1.1.1.5.6.2 case

Category

Statements

Syntax

```
switch ( <switch variable> ){casebreakdefault
case <constant expression> : <statement>; [break;]
    .
    .
    .
default : <statement>;
}
```

Description

Use the **case** statement in conjunction with switches to determine which statements evaluate.

The list of possible branch points within <statement> is determined by preceding substatements with

```
case <constant expression> : <statement>;
```

where <constant expression> must be an **int** and must be unique.

The <constant expression> values are searched for a match for the <switch variable>.

If a match is found, execution continues after the matching **case** statement until a **break** statement is encountered or the end of the **switch** statement is reached.

If no match is found, control is passed to the **default** case.

Note: It is illegal to have duplicate **case** constants in the same **switch** statement.

See Also

Break (see page 191)

Default (see page 192)

1.1.1.5.6.3 continue

Category

Statements

Syntax

```
continue;
```

Description

Use the **continue** statement within loops to pass control to the end of the innermost enclosing end brace belonging to a looping construct, such as for or while; at which point the loop continuation condition is re-evaluated.

See Also

While (see page 194)

Do (see page 192)

1.1.1.5.6.4 default

Category

Statements

Syntax

```
switch ( <switch variable> ){casebreakdefault
case <constant expression> : <statement>; [break;]
     .
     .
     .
default : <statement>;
}
```

Description

Use the default statement in switch statement blocks.

- If a case match is not found and the **default** statement is found within the switch statement, the execution continues at this point.
- If no default is defined in the **switch** statement, control passes to the next statement that follows the switch statement block.

1.1.1.5.6.5 do

Category

Statements

Syntax

```
do <statement> while ( <condition> );
```

Description

The **do** statement executes until the condition becomes **false**.

<statement> is executed repeatedly as long as the value of <condition> remains **true**.

Since the conditon tests after each the loop executes the <statement>, the loop will execute at least once.

See Also

While (see page 194)

1.1.1.5.6.6 for

Category

Statements

Syntax

```
for ( [<initialization>] ; [<condition>] ; [<increment>] )  <statement>
```

Description

The **for** statement implements an iterative loop.

<condition> is checked before the first entry into the block.

<statement> is executed repeatedly UNTIL the value of <condition> is false.

- Before the first iteration of the loop, <initialization> initializes variables for the loop.
- After each iteration of the loop, <increments> increments a loop counter. Consequently, j++ is functionally the same as ++j.

In C++, <initialization> can be an expression or a declaration.

The scope of any identifier declared within the **for** loop extends to the end of the control statement only.

A variable defined in the **for**-initialization expression is in scope only within the **for**-block. See the description of the -Vd option.

All the expressions are optional. If <condition> is left out, it is assumed to be always true.

1.1.1.5.6.7 goto

Category

Statements

Syntax

```
goto <identifier> ;
```

Description

Use the **goto** statement to transfer control to the location of a local label specified by <identifier>.

Labels are always terminated by a colon.

1.1.1.5.6.8 return

Category

Statements

Syntax

```
return [ <expression> ] ;
```

Description

Use the **return** statement to exit from the current function back to the calling routine, optionally returning a value.

1.1.1.5.6.9 switch

Category

Statements

Syntax

```
switch ( <switch variable> ) {casebreakdefault
case <constant expression> : <statement>; [break;]
    .
    .
    .
default : <statement>;
}
```

Description

Use the switch statement to pass control to a case that matches the <switch variable>. At which point the statements following the matching case evaluate.

If no case satisfies the condition the default case evaluates.

To avoid evaluating any other cases and relinquish control from the switch, terminate each case with **break**.

See Also

Break (see page 191)

Case (see page 191)

1.1.1.5.6.10 while

Category

Statements

Syntax

```
while ( <condition> ) <statement>
```

Description

Use the **while** keyword to conditionally iterate a statement.

<statement> executes repeatedly until the value of <condition> is **false**.

The test takes place before <statement> executes. Thus, if <condition> evaluates to **false** on the first pass, the loop does not execute.

1.1.1.5.7 Storage Class Specifiers

This section contains Storage Class Specifier topics.

1.1.1.5.7.1 auto

Category

Storage class specifiers

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.

1.1.1.5.7.2 extern

Category

Storage class specifiers

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.

1.1.1.5.7.3 register

Category

Storage class specifiers

Syntax

```
register <data definition> ;
```

Description

Use the register storage class specifier to store the variable being declared in a CPU register (if possible), to optimize access and reduce code.

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

1.1.1.5.7.4 typedef

Category

Storage class specifiers

Syntax

```
typedef <type definition> <identifier> ;
```

Description

Use the **typedef** keyword to assign the symbol name <identifier> to the data type definition <type definition>.

1.1.1.5.8 Type Specifiers

This section contains Type Specifier topics.

1.1.1.5.8.1 char

Category

Type specifiers

Syntax

```
[signed|unsigned] char <variable_name>
```

Description

Use the type specifier **char** to define a character data type. Variables of type **char** are 1 byte in length.

A **char** can be signed, unsigned, or unspecified. By default, **signed char** is assumed.

Objects declared as characters (**char**) are large enough to store any member of the basic ASCII character set.

1.1.1.5.8.2 double

Category

Type specifiers

Syntax

```
[long] double <identifier>
```

Description

Use the **double** type specifier to define an identifier to be a floating-point data type. The optional modifier **long** extends the accuracy of the floating-point value.

If you use the **double** keyword, the floating-point math package will automatically be linked into your program.

See Also

Float (see page 197)

Long (see page 198)

1.1.1.5.8.3 enum

Category

Type specifiers

Syntax

```
enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];
```

- <type_tag> is an optional type tag that names the set.
- <constant_name> is the name of a constant that can optionally be assigned the value of <value>. These are also called enumeration constants.
- <value> must be an integer. If <value> is missing, it is assumed to be: <prev> + 1 where <prev> is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0.
- <var_list> is an optional variable list that assigns variables to the enum type.

Description

Use the **enum** keyword to define a set of constants of type **int**, called an enumeration data type.

An enumeration data type provides mnemonic identifiers for a set of integer values. Use the **-b** flag to toggle the Treat Enums As Ints option. .Enums are always interpreted as **int**s if the range of values permits this, 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**, **short**, **unsigned short**, **int**, or **unsigned int**.

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

In C++, you can omit the **enum** keyword if <tag_type> is not the name of anything else in the same scope. You can also omit <tag_type> if no further variables of this **enum** type are required.

In the absence of a <value> the first enumerator is assigned the value of zero. Any subsequent names without initializers will then increase by one. <value> can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

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

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

1.1.1.5.8.4 float

Category

Type specifiers

Syntax

```
float <identifier>
```

Description

Use the **float** type specifier to define an identifier to be a floating-point data type.

Type	Length	Range
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)

The floating-point math package will be automatically linked into your program if you use floating-point values or operators.

See Also

Double (see page 196)

1.1.1.5.8.5 int

Category

Type specifiers

Syntax

```
[signed|unsigned] int <identifier> ;
```

Description

Use the **int** type specifier to define an integer data type.

Variables of type **int** can be signed (default) or unsigned.

1.1.1.5.8.6 long

Category

Type specifiers

Syntax

```
long [int] <identifier> ;
[long] double <identifier> ;
```

Description

When used to modify a **double**, it defines a floating-point data type with 80 bits of precision instead of 64.

The floating-point math package will be automatically linked with your program if you use floating-point values or operators.

1.1.1.5.8.7 short

Category

Type specifiers

Syntax

```
short int <variable> ;
```

Description

Use the short type modifier when you want a variable smaller than an int. This modifier can be applied to the base type int.

When the base type is omitted from a declaration, int is assumed.

See Also

Long (see page 198)

Signed (see page 198)

1.1.1.5.8.8 signed

Category

Type specifiers

Syntax

```
signed <type> <variable> ;
```

Description

Use the signed type modifier when the variable value can be either positive or negative. The signed modifier can be applied to base types **int**, **char**, **long**, **short**, and **__int64**.

When the base type is omitted from a declaration, **int** is assumed.

See Also

Char (see page 196)

Int (see page 197)

Long (see page 198)

Short (see page 198)

Unsigned (see page 200)

1.1.1.5.8.9 struct

Category

Type specifiers

Syntax

```
struct [<struct type name>] {
[<type> <variable-name[, variable-name, ...]>] ;
    .
    .
    .
} [<structure variables>] ;
```

Description

Use a **struct** to group variables into a single record.

<struct type name> An optional tag name that refers to the structure type.

<structure variables> The data definitions, also optional.

Though both <struct type name> and <structure variables> are optional, one of the two must appear.

You define elements in the record by naming a <type>, followed by one or more <variable-name> (separated by commas).

Separate different variable types by a semicolon.

Use the **.** operator, or the **->** operator to access elements in a structure.

To declare additional variables of the same type, use the keyword **struct** followed by the <struct type name>, followed by the variable names. In C++ the keyword **struct** can be omitted.

Note: The compiler allows the use of anonymous struct embedded within another structure.

See Also

Class (see page 170)

Public (see page 174)

Union (see page 199)

1.1.1.5.8.10 union

Category

Type specifiers

Syntax

```
union [<union type name>] {
<type> <variable names> ;
  ...
} [<union variables>] ;
```

Description

Use unions to define variables that share storage space.

The compiler allocates enough storage in a_number to accommodate the largest element in the union.

Unlike a struct, the members of a union occupy the same location in memory. Writing into one overwrites all others.

Use the record selector (.) to access elements of a union .

See Also

Class (see page 170)

Public (see page 174)

1.1.1.5.8.11 unsigned

Category

Type specifiers

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, short,** and **__int64**.

When the base type is omitted from a declaration, **int** is assumed.

See Also

Char (see page 196)

Int (see page 197)

Long (see page 198)

Short (see page 198)

Signed (see page 198)

1.1.2 C Runtime Library Reference

RAD Studio has several hundred classes, functions, and macros that you call from within your C and C++ programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and more.

1.1.2.1 alloc.h

The following functions, macros, and classes are provided in `alloc.h`:

1.1.2.1.1 calloc

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void *calloc(size_t nitems, size_t size);
```

Description

Allocates main memory.

calloc provides access to the C memory heap. The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

calloc allocates a block of size nitems * size. The block is initialized to 0.

Return Value

calloc returns a pointer to the newly allocated block. If not enough space exists for the new block or if nitems or size is 0, calloc returns NULL.

Example

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *str = NULL;
   /* allocate memory for string */
   str = (char *) calloc(10, sizeof(char));
   /* copy "Hello" into string */
   strcpy(str, "Hello");
   /* display string */
   printf("String is %s\n", str);
   /* free memory */
   free(str);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.1.2 free

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void free(void *block);
```

Description

Frees allocated block.

free deallocates a memory block allocated by a previous call to calloc, malloc, or realloc.

Return Value

None.

Example

```
#include <string.h>
#include <stdio.h>
#include <alloc.h>
int main(void)
{
   char *str;
   /* allocate memory for string */
   str = (char *) malloc(10);
   /* copy "Hello" to string */
   strcpy(str, "Hello");
   /* display string */
   printf("String is %s\n", str);
   /* free memory */
   free(str);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.1.3 heapcheck

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapcheck(void);
```

Description

Checks and verifies the heap.

heapcheck walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

Return Value

The return value is less than 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is verified

Example

```
#include <stdio.h>
#include <alloc.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
```

```
int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
   if( heapcheck() == _HEAPCORRUPT )
      printf( "Heap is corrupted.\n" );
   else
      printf( "Heap is OK.\n" );
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.4 heapcheckfree

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapcheckfree(unsigned int fillvalue);
```

Description

Checks the free blocks on the heap for a constant value.

Return Value

The return value is less then 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

_BADVALUE	A value other than the fill value was found
_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is accurate

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.5 heapchecknode

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapchecknode(void *node);
```

Description

Checks and verifies a single node on the heap.

If a node has been freed and heapchecknode is called with a pointer to the freed block, heapchecknode can return _BADNODE rather than the expected _FREEENTRY. This is because adjacent free blocks on the heap are merged, and the block in question no longer exists.

Return Value

One of the following values:

_BADNODE	Node could not be found
_FREEENTRY	Node is a free block
_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_USEDENTRY	Node is a used block

Example

```
#include <stdio.h>
#include <alloc.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
   for( i = 0; i < NUM_PTRS; i++ )
   {
      printf( "Node %2d ", i );
      switch( heapchecknode( array[ i ] ) )
      {
         case _HEAPEMPTY:
              printf( "No heap.\n" );
              break;
         case _HEAPCORRUPT:
              printf( "Heap corrupt.\n" );
              break;
         case _BADNODE:
              printf( "Bad node.\n" );
              break;
         case _FREEENTRY:
              printf( "Free entry.\n" );
              break;
         case _USEDENTRY:
              printf( "Used entry.\n" );
              break;
         default:
              printf( "Unknown return code.\n" );
               break;
```

```
      }
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.6 _heapchk

Header File

malloc.h

Category

Memory Routines

Syntax

```
int _heapchk(void);
```

Description

Checks and verifies the heap.

_heapchk walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

Return Value

One of the following values:

_HEAPBADNODE	A corrupted heap block has been found
_HEAPEMPTY	No heap exists
_HEAPOK	The heap appears to be uncorrupted

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.7 heapfillfree

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapfillfree(unsigned int fillvalue);
```

Description

Fills the free blocks on the heap with a constant value.

Return Value

One of the following values:

_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is accurate

Example

```
#include <stdio.h>
#include <alloc.h>
#include <mem.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   int res;
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
   if( heapfillfree( 1 ) < 0 )
   {
      printf( "Heap corrupted.\n" );
      return 1;
   }
   for( i = 1; i < NUM_PTRS; i += 2 )
      memset( array[ i ], 0, NUM_BYTES );
   res = heapcheckfree( 1 );
   if( res < 0 )
      switch( res )
      {
         case _HEAPCORRUPT:
              printf( "Heap corrupted.\n" );
              return 1;
         case _BADVALUE:
              printf( "Bad value in free space.\n" );
              return 1;
         default:
              printf( "Unknown error.\n" );
              return 1;
      }
   printf( "Test successful.\n" );
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.8 _heapmin

Header File

malloc.h

Category

Memory Routines

Prototype

```
int _heapmin(void);
```

Description

Release unused heap areas.

The _heapmin function returns unused areas of the heap to the operating system. This allows other processes to use blocks that have been allocated and then freed. Due to fragmentation of the heap, _heapmin might not always be able to return unused memory to the operating system; this is not an error.

Return Value

_heapmin returns 0 if it is successful, or -1 if an error occurs.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.9 _heapset

Header File

malloc.h

Category

Memory Routines

Prototype

```
int _heapset(unsigned int fillvalue);
```

Description

Fills the free blocks on the heap with a constant value.

_heapset checks the heap for consistency using the same methods as _heapchk. It then fills each free block in the heap with the value contained in the least significant byte of fillvalue. This function can be used to find heap-related problems. It does not guarantee that subsequently allocated blocks will be filled with the specified value.

Return Value

One of the following values:

_HEAPOK	The heap appears to be uncorrupted
_HEAPEMPTY	No heap exists
_HEAPBADNODE	A corrupted heap block has been found

Example

```
#include <windowsx.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
```

```
BOOL InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam);
void ExampleHeapSet(HWND hWnd);
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;      // message
  if (!InitApplication(hInstance))  // Initialize shared things
   return (FALSE); // Exits if unable to initialize
  /* Perform initializations that apply to a specific instance */
  if (!(InitInstance(hInstance, nCmdShow)))
   return (FALSE);
  /* Acquire and dispatch messages until a WM_QUIT message is received. */
  while (GetMessage(&msg, // message structure
   NULL, // handle of window receiving the message
   NULL, // lowest message to examine
   NULL))  // highest message to examine
  {
  TranslateMessage(&msg); // Translates virtual key codes
  DispatchMessage(&msg);  // Dispatches message to window
  }
  return (msg.wParam);  // Returns the value from PostQuitMessage
}
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;
  // Fill in window class structure with parameters that describe the
  // main window.
  wc.style = CS_HREDRAW | CS_VREDRAW; // Class style(s).
  wc.lpfnWndProc = (long (FAR PASCAL*)(void *,unsigned int,unsigned int, long ))MainWndProc;
// Function to retrieve messages for
                // windows of this class.
  wc.cbClsExtra = 0;  // No per-class extra data.
  wc.cbWndExtra = 0;  // No per-window extra data.
  wc.hInstance = hInstance; // Application that owns the class.
  wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = NULL;  // Name of menu resource in .RC file.
  wc.lpszClassName = "Example"; // Name used in call to CreateWindow.
  /* Register the window class and return success/failure code. */
  return (RegisterClass(&wc));
}
HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND  hWnd; // Main window handle.
  /* Create a main window for this application instance.  */
  hWnd = CreateWindow(
   "Example",                 // See RegisterClass() call.
   "Example _heapset 32 bit only",  // Text for window title bar.
   WS_OVERLAPPEDWINDOW,        // Window style.
   CW_USEDEFAULT,         // Default horizontal position.
   CW_USEDEFAULT,          // Default vertical position.
   CW_USEDEFAULT,            // Default width.
   CW_USEDEFAULT,        // Default height.
   NULL,                  // Overlapped windows have no parent.
   NULL,                  // Use the window class menu.
   hInstance,              // This instance owns this window.
   NULL                   // Pointer not needed.
  );
  /* If window could not be created, return "failure" */
  if (!hWnd)
   return (FALSE);
```

```
  /* Make the window visible; update its client area; and return "success" */
  ShowWindow(hWnd, nCmdShow); // Show the window
  UpdateWindow(hWnd);     // Sends WM_PAINT message
  return (hWnd);        // Returns the value from PostQuitMessage
}
void ExampleHeapSet(HWND hWnd)
{
  int hsts;
  char *buffer;
  if ( (buffer = (char *)malloc( 1 )) == NULL )
    exit(0);
  hsts = _heapset( 'Z' );
  switch (hsts)
  {
    case _HEAPOK:
      MessageBox(hWnd,"Heap is OK","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    case _HEAPEMPTY:
      MessageBox(hWnd,"Heap is empty","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    case _HEAPBADNODE:
      MessageBox(hWnd,"Bad node in heap","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    default:
      break;
  }
  free (buffer);
}
#pragma argsused
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    case WM_CREATE:
    {
      //Example _heapset
      ExampleHeapSet(hWnd);
      return NULL;
    }
    case WM_QUIT:
    case WM_DESTROY:  // message: window being destroyed
      PostQuitMessage(0);
    break;
    default:      // Passes it on if unprocessed
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.10 heapwalk

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapwalk(struct heapinfo *hi);
```

Description

heapwalk is used to “walk” through the heap, node by node.

heapwalk assumes the heap is correct. Use heapcheck to verify the heap before using heapwalk. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to heapwalk.

heapwalk receives a pointer to a structure of type heapinfo (declared in alloc.h). For the first call to heapwalk, set the hi.ptr field to null. heapwalk returns with hi.ptr containing the address of the first block. hi.size holds the size of the block in bytes. hi.in_use is a flag that’s set if the block is currently in use.

Return Value

One of the following values:

_HEAPEMPTY	No heap exists
_HEAPEND	The end of the heap has been reached
_HEAPOK	The heapinfo block contains valid information about the next heap block

Example

```
Portability
<table htmltable<tr><th>POSIX<th>Win32<th>ANSI C<th>ANSI C++<tr><td> <td>+<td> <td> </table
```

1.1.2.1.11 malloc

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void *malloc(size_t size);
```

Description

malloc allocates a block of size bytes from the memory heap. It allows a program to allocate memory explicitly as it’s needed, and in the exact amounts needed.

Allocates main memory.The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ heap memory allocation.

In the large data models, all the space beyond the program stack to the end of available memory is available for the heap.

Return Value

On success, malloc returns a pointer to the newly allocated block of memory. If not enough space exists for the new block, it returns NULL. The contents of the block are left unchanged. If the argument size == 0, malloc returns NULL.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
```

```
int main(void)
{
    char *str;
    /* allocate memory for string */
    if ((str = (char *) malloc(10)) == NULL)
    {
       printf("Not enough memory to allocate buffer\n");
       exit(1);  /* terminate program if out of memory */
    }
    /* copy "Hello" into string */
    strcpy(str, "Hello");
    /* display string */
    printf("String is %s\n", str);
    /* free memory */
    free(str);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.1.12 _msize

Header File

malloc.h

Category

Memory Routines

Prototype

```
size_t _msize(void *block);
```

Description

Returns the size of a heap block.

_msize returns the size of the allocated heap block whose address is block. The block must have been allocated with malloc, calloc, or realloc. The returned size can be larger than the number of bytes originally requested when the block was allocated.

Return Value

_msize returns the size of the block in bytes.

Example

```
#include <malloc.h>        /* malloc() _msize() */
#include <stdio.h>            /* printf() */
int main( )
{
  int size;
  int *buffer;
  buffer = malloc(100 * sizeof(int));
  size = _msize(buffer);
  printf("Allocated %d bytes for 100 integers\n", size);
  return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.13 realloc

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void *realloc(void *block, size_t size);
```

Description

Reallocates main memory.

realloc attempts to shrink or expand the previously allocated block to size bytes. If size is zero, the memory block is freed and NULL is returned. The block argument points to a memory block previously obtained by calling malloc, calloc, or realloc. If block is a NULL pointer, realloc works just like malloc.

realloc adjusts the size of the allocated block to size, copying the contents to a new location if necessary.

Return Value

realloc returns the address of the reallocated block, which can be different than the address of the original block.

If the block cannot be reallocated, realloc returns NULL.

If the value of size is 0, the memory block is freed and realloc returns NULL.

Example

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *str;
   /* allocate memory for string */
   str = (char *) malloc(10);
   /* copy "Hello" into string */
   strcpy(str, "Hello");
   printf("String is %s\n  Address is %p\n", str, str);
   str = (char *) realloc(str, 20);
   printf("String is %s\n  New address is %p\n", str, str);
   /* free memory */
   free(str);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.1.14 _rtl_heapwalk

Header File

malloc.h

Category

Memory Routines

Prototype

```
int _rtl_heapwalk(_HEAPINFO *hi);
```

Description

Inspects the heap node by node.

Note: This function replaces _heapwalk which is obsolete.

_rtl_heapwalk assumes the heap is correct. Use _heapchk to verify the heap before using _rtl_heapwalk. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to _rtl_heapwalk.

_rtl_heapwalk receives a pointer to a structure of type _HEAPINFO (declared in malloc.h). Note that the _HEAPINFO structure must be allocated on the heap (using malloc()). You can't pass the address of a variable declared on the stack.

For the first call to _rtl_heapwalk, set the hi._pentry field to NULL. _rtl_heapwalk returns with hi._pentry containing the address of the first block.

hi._size	holds the size of the block in bytes.
hi._useflag	is a flag that is set to _USEDENTRY if the block is currently in use. If the block is free, hi._useflag is set to _FREEENTRY.

Return Value

This function returns one of the following values:

_HEAPBADNODE	A corrupted heap block has been found
_HEAPBADPTR	The _pentry field does not point to a valid heap block
_HEAPEMPTY	No heap exists
_HEAPEND	The end of the heap has been reached
_HEAPOK	The _heapinfo block contains valid information about the next heap block

Example

```
#include <stdio.h>
#include <malloc.h>
#include <alloc.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main( void )
{
   _HEAPINFO *hi;
   char *array[ NUM_PTRS ];
   int i;
   hi = (_HEAPINFO *) malloc( sizeof(_HEAPINFO) );
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
```

```
    for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
    hi->_pentry = NULL;
    printf( "   Size   Status\n" );
    printf( "   ----   ------\n" );
    while( _rtl_heapwalk( hi ) == _HEAPOK )
      printf( "%7u    %s\n", hi->_size, hi->_useflag ? "used" : "free" );
    free( hi );
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.1.15 stackavail

Header File

malloc.h

Category

Memory Routines

Prototype

```
size_t stackavail(void);
```

Description

Gets the amount of available stack memory.

stackavail returns the number of bytes available on the stack. This is the amount of dynamic memory that alloca can access.

Return Value

stackavail returns a size_t value indicating the number of bytes available.

Example

```
#include <malloc.h>
#include <stdio.h>

int main(void)
{
  char *buf;

  printf("\nThe stack: %u\tstack pointer: %u", stackavail(), _SP);
  buf = (char *) alloca(100 * sizeof(char));
  printf("\nNow, the stack: %u\tstack pointer: %u", stackavail(), _SP);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.2 assert.h

The following functions, macros, and classes are provided in `assert.h`:

1.1.2.2.1 assert

Header File

assert.h

Category

Diagnostic Routines

Prototype

```
void assert(int test);
```

Description

Tests a condition and possibly aborts.

assert is a macro that expands to an **if** statement; if test evaluates to zero, the assert macro calls the _assert function

```
void _RTLENTRY _EXPFUNC _assert(char * __cond, char * __file, int __line);
```

and aborts the program. The _assert function calls abort and asserts the following a message on stderr:

```
Assertion failed: test, file filename, line linenum
```

The filename and linenum listed in the message are the source file name and line number where the assert macro appears.

If you place the #define NDEBUG directive ("no debugging") in the source code before the #include <assert.h> directive, the macro expands to a no-op, the effect is to comment out the assert statement.

Return Value

None.

Example

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
struct ITEM {
   int key;
   int value;
};
/* add item to list, make sure list is not  null */
void additem(struct ITEM *itemptr) {
   assert(itemptr != NULL);
   /* add item to list */
}
int main(void)
{
   additem(NULL);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.2.2 NDEBUG #define

Header File

assert.h

Description

NDEBUG means "Use #define to treat assert as a macro or a true function".

Can be defined in a user program. If defined, assert is a true function; otherwise assert is a macro.

1.1.2.3 conio.h

The following functions, macros, and classes are provided in `conio.h`:

1.1.2.3.1 cgets

Header File

conio.h

Category

Console I/O Routines

Prototype

```
char *cgets(char *str);
```

Description

Reads a string from the console.

cgets reads a string of characters from the console, storing the string (and the string length) in the location pointed to by str.

cgets reads characters until it encounters a carriage-return/linefeed (CR/LF) combination, or until the maximum allowable number of characters have been read. If cgets reads a CR/LF combination, it replaces the combination with a \0 (null terminator) before storing the string.

Before cgets is called, set str[0] to the maximum length of the string to be read. On return, str[1] is set to the number of characters actually read. The characters read start at str[2] and end with a null terminator. Thus, str must be at least str[0] plus 2 bytes long.

Note: Do not use this function for Win32 GUI applications.

Return Value

On success, cgets returns a pointer to str[2].

Example

```
#include <stdio.h>
```

```
#include <conio.h>
int main(void)
{
   char buffer[83];
   char *p;
   /* There is space for 80 characters plus the NULL terminator */
   buffer[0] = 81;
   printf("Input some chars:");
   p = cgets(buffer);
   printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
   printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
   /* Leave room for 5 characters plus the NULL terminator */
   buffer[0] = 6;
   printf("Input some chars:");
   p = cgets(buffer);
   printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
   printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.2 clreol

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void clreol(void);
```

Description

Clears to end of line in text window.

clreol clears all characters from the cursor position to the end of the line within the current text window, without moving the cursor.

Note: This function should not be used in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   clrscr();
   cprintf("The function CLREOL clears all characters from the\r\n");
   cprintf("cursor position to the end of the line within the\r\n");
   cprintf("current text window, without moving the cursor.\r\n");
   cprintf("Press any key to continue . . .");
   gotoxy(14, 4);
```

```
   getch();
   clreol();
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.3 clrscr

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void clrscr(void);
```

Description

Clears the text-mode window.

clrscr clears the current text window and places the cursor in the upper left corner (at position 1,1).

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   int i;
   clrscr();
   for (i = 0; i < 20; i++)
      cprintf("%d\r\n", i);
   cprintf("\r\nPress any key to clear screen");
   getch();
   clrscr();
   cprintf("The screen has been cleared!");
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.4 cprintf

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int cprintf(const char *format[, argument, ...]);
```

Description

Writes formatted output to the screen.

cprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data directly to the current text window on the screen. There must be the same number of format specifiers as arguments.

For details details on format specifiers, see printf Format Specifiers.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo.

Unlike fprintf and printf, cprintf does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n). Tab characters (specified by \t) are not expanded into spaces.

Note: Do not use this function in Win32 GUI applications.

Return Value

cprintf returns the number of characters output.

Example

```
#include <conio.h>
int main(void)
{
   /* clear the screen */
   clrscr();
   /* create a text window */
   window(10, 10, 80, 25);
   /* output some text in the window */
   cprintf("Hello world\r\n");
   /* wait for a key */
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.5 cputs

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int cputs(const char *str);
```

Description

Writes a string to the screen.

cputs writes the null-terminated string str to the current text window. It does not append a newline character.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo. Unlike puts, cputs does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n).

Note: Do not use this function in Win32 GUI applications.

Return Value

cputs returns the last character printed.

Example

```
#include <conio.h>
int main(void)
{
   /* clear the screen */
   clrscr();
   /* create a text window */
   window(10, 10, 80, 25);
   /* output some text in the window */
   cputs("This is within the window\r\n");
   /* wait for a key */
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.6 cscanf

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int cscanf(char *format[, address, ...]);
```

Description

Scans and formats input from the console.

cscanf scans a series of input fields one character at a time, reading directly from the console. Then each field is formatted according to a format specifier passed to cscanf in the format string pointed to by format. Finally, cscanf stores the formatted input at an address passed to it as an argument following format, and echoes the input directly to the screen. There must be the same number of format specifiers and addresses as there are input fields.

Note: For details on format specifiers, see scanf Format Specifiers.

cscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely for a number of reasons. See scanf for a discussion of possible causes.

Note: Do not use this function in Win32 GUI applications.

Return Value

cscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If cscanf attempts to read at end-of-file , the return value is EOF.

Example

```
#include <conio.h>
int main(void)
{
   char string[80];
   /* clear the screen */
   clrscr();
   /* Prompt the user for input */
   cprintf("Enter a string with no spaces:");
   /* read the input */
   cscanf("%s", string);
   /* display what was read */
   cprintf("\r\nThe string entered is: %s", string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.7 delline

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void delline(void);
```

Description

Deletes line in text window.

delline deletes the line containing the cursor and moves all lines below it one line up. delline operates within the currently active text window.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   clrscr();
   cprintf("The function DELLINE deletes the line containing the\r\n");
   cprintf("cursor and moves all lines below it one line up.\r\n");
   cprintf("DELLINE operates within the currently active text\r\n");
   cprintf("window.  Press any key to continue . . .");
   gotoxy(1,2);  /* Move the cursor to the second line and first column */
   getch();
   delline();
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.8 getch

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int getch(void);
```

Description

Gets character from keyboard, does not echo to screen.

getch reads a single character directly from the keyboard, without echoing to the screen.

Note: Do not use this function in Win32 GUI applications.

Return Value

getch returns the character read from the keyboard.

Example

```
#include <conio.h>
#include <stdio.h>
int main(void)
{
  int c;
  int extended = 0;
  c = getch();
```

```
  if (!c)
    extended = getch();
  if (extended)
    printf("The character is extended\n");
  else
    printf("The character isn't extended\n");
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.9 getche

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int getche(void);
```

Description

Gets character from the keyboard, echoes to screen.

getche reads a single character from the keyboard and echoes it to the current text window using direct video or BIOS.

Note: Do not use this function in Win32 GUI applications.

Return Value

getche returns the character read from the keyboard.

Example

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
   char ch;
   printf("Input a character:");
   ch = getche();
   printf("\nYou input a '%c'\n", ch);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.10 getpass

Header File

conio.h

Category

Console I/O Routines

Prototype

```
char *getpass(const char *prompt);
```

Description

Reads a password.

getpass reads a password from the system console after prompting with the null-terminated string prompt and disabling the echo. A pointer is returned to a null-terminated string of up to eight characters (not counting the null-terminator).

Note: Do not use this function in Win32 GUI applications.

Return Value

The return value is a pointer to a static string that is overwritten with each call.

Example

```
#include <conio.h>

int main(void)
{
   char *password;

   password = getpass("Input a password:");
   cprintf("The password is: %s\r\n", password);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.11 gettext

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int gettext(int left, int top, int right, int bottom, void *destin);
```

Description

Copies text from text mode screen to memory.

gettext stores the contents of an onscreen text rectangle defined by left, top, right, and bottom into the area of memory pointed to by destin.

All coordinates are absolute screen coordinates not window-relative. The upper left corner is (1,1). gettext reads the contents of the rectangle into memory sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell and the second is the cell's video attribute. The space required for a rectangle w columns wide by h rows high is defined as

bytes = (h rows) x (w columns) x 2

Note: Do not use this function in Win32 GUI applications.

Return Value

gettext returns 1 if the operation succeeds.

On error, it returns 0 (for example, if it fails because you gave coordinates outside the range of the current screen mode).

Example

```
#include <conio.h>

char buffer[4096];
int main(void)
{
   int i;
   clrscr();
   for (i = 0; i <= 20; i++)
      cprintf("Line #%d\r\n", i);
   gettext(1, 1, 80, 25, buffer);
   gotoxy(1, 25);
   cprintf("Press any key to clear screen...");
   getch();
   clrscr();
   gotoxy(1, 25);
   cprintf("Press any key to restore screen...");
   getch();
   puttext(1, 1, 80, 25, buffer);
   gotoxy(1, 25);
   cprintf("Press any key to quit...");
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.12 gettextinfo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void gettextinfo(struct text_info *r);
```

Description

Gets text mode video information.

gettextinfo fills in the text_info structure pointed to by r with the current text video information.

The text_info structure is defined in conio.h as follows:

```
struct text_info {
unsigned char winleft; /* left window coordinate */
unsigned char wintop; /* top window coordinate */
unsigned char winright; /* right window coordinate */
unsigned char winbottom; /* bottom window coordinate */
unsigned char attribute; /* text attribute */
unsigned char normattr; /* normal attribute */
unsigned char currmode; /* BW40, BW80, C40, C80, or C4350 */
unsigned char screenheight; /* text screen's height */
unsigned char screenwidth; /* text screen's width */
unsigned char curx; /* x-coordinate in current window */
unsigned char cury; /* y-coordinate in current window */
};
```

Note: Do not use this function in Win32 GUI applications.

Return Value

None. Results are returned in the structure pointed to by r.

Example

```
#include <conio.h>
int main(void)
 {
   struct text_info ti;
   gettextinfo(&ti);
   cprintf("window left      %2d\r\n",ti.winleft);
   cprintf("window top       %2d\r\n",ti.wintop);
   cprintf("window right     %2d\r\n",ti.winright);
   cprintf("window bottom    %2d\r\n",ti.winbottom);
   cprintf("attribute        %2d\r\n",ti.attribute);
   cprintf("normal attribute %2d\r\n",ti.normattr);
   cprintf("current mode     %2d\r\n",ti.currmode);
   cprintf("screen height    %2d\r\n",ti.screenheight);
   cprintf("screen width     %2d\r\n",ti.screenwidth);
   cprintf("current x        %2d\r\n",ti.curx);
   cprintf("current y        %2d\r\n",ti.cury);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.13 gotoxy

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void gotoxy(int x, int y);
```

Description

Positions cursor in text window.

gotoxy moves the cursor to the given position in the current text window. If the coordinates are in any way invalid the call to gotoxy is ignored. An example of this is a call to gotoxy(40,30) when (35,25) is the bottom right position in the window. Neither argument to gotoxy can be zero.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
    clrscr();
    gotoxy(35, 12);
    cprintf("Hello world");
    getch();
    return 0;
}
```

1.1.2.3.14 highvideo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void highvideo(void);
```

Description

Selects high-intensity characters.

highvideo selects high-intensity characters by setting the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen, but does affect those displayed by functions (such as cprintf) that perform direct video, text mode output after highvideo is called.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
clrscr();
lowvideo();
cprintf("Low Intensity text\r\n");
highvideo();
gotoxy(1,2);
cprintf("High Intensity Text\r\n");
return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.15 insline

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void insline(void);
```

Description

Inserts a blank line in the text window.

insline inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line, and the bottom line scrolls off the bottom of the window.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
    clrscr();
    cprintf("INSLINE inserts an empty line in the text window\r\n");
 cprintf("at the cursor position using the current text\r\n");
    cprintf("background color.  All lines below the empty one\r\n");
    cprintf("move down one line and the bottom line scrolls\r\n");
 cprintf("off the bottom of the window.\r\n");
    cprintf("\r\nPress any key to continue:");
 gotoxy(1, 3);
```

```
 getch();
    insline();
    getch();
 return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.16 kbhit

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int kbhit(void);
```

Description

Checks for currently available keystrokes.

kbhit checks to see if a keystroke is currently available. Any available keystrokes can be retrieved with getch or getche.

Note: Do not use this function in Win32 GUI applications.

Return Value

If a keystroke is available, kbhit returns a nonzero value. Otherwise, it returns 0.

Example

```
#include <conio.h>
int main(void)
{
 cprintf("Press any key to continue:");
    while (!kbhit()) /* do nothing */ ;
    cprintf("\r\nA key was pressed...\r\n");
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.17 lowvideo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void lowvideo(void);
```

Description

Selects low-intensity characters.

lowvideo selects low-intensity characters by clearing the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen. It affects only those characters displayed by functions that perform text mode, direct console output after this function is called.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
clrscr();
highvideo();
   cprintf("High Intensity Text\r\n");
   lowvideo();
gotoxy(1,2);
   cprintf("Low Intensity Text\r\n");
return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.18 movetext

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int movetext(int left, int top, int right, int bottom, int destleft, int desttop);
```

Description

Copies text onscreen from one rectangle to another.

movetext copies the contents of the onscreen rectangle defined by left, top, right, and bottom to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (destleft, desttop).

All coordinates are absolute screen coordinates. Rectangles that overlap are moved correctly.

movetext is a text mode function performing direct video output.

Note: Do not use this function in Win32 GUI applications.

Return Value

On success, movetext returns nonzero.

On error (for example, if it failed because you gave coordinates outside the range of the current screen mode), movetext returns 0.

Example

```
#include <conio.h>
#include <string.h>
int main(void)
{
    char *str = "This is a test string";
    clrscr();
    cputs(str);
    getch();
    movetext(1, 1, strlen(str), 2, 10, 10);
    getch();
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.19 normvideo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void normvideo(void);
```

Description

Selects normal-intensity characters.

normvideo selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started.

This function does not affect any characters currently on the screen, only those displayed by functions (such as cprintf) performing direct console output functions after normvideo is called.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
    normvideo();
    cprintf("NORMAL Intensity Text\r\n");
 return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.20 putch

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int putch(int c);
```

Description

Outputs character to screen.

putch outputs the character c to the current text window. It is a text mode function performing direct video output to the console. putch does not translate linefeed characters (\n) into carriage-return/linefeed pairs.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo.

Note: This function should not be used in Win32 GUI applications.

Return Value

On success, putch returns the character printed, c. On error, it returns EOF.

Example

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
  char ch = 0;
  printf("Input a string:");
  while ((ch != '\r'))
  {
    ch = getch();
    putch(ch);
  }
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.21 puttext

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int puttext(int left, int top, int right, int bottom, void *source);
```

Description

Copies text from memory to the text mode screen.

puttext writes the contents of the memory area pointed to by source out to the onscreen rectangle defined by left, top, right, and bottom.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

puttext places the contents of a memory area into the defined rectangle sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the cell's video attribute. The space required for a rectangle w columns wide by h rows high is defined as

bytes = (h rows) x (w columns) x 2

puttext is a text mode function performing direct video output.

Note: This function should not be used in Win32 GUI applications.

Return Value

puttext returns a nonzero value if the operation succeeds; it returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

Example

```
#include <conio.h>
int main(void)
{
   char buffer[512];
   /* put some text to the console */
   clrscr();
   gotoxy(20, 12);
   cprintf("This is a test.  Press any key to continue ...");
   getch();
   /* grab screen contents */
   gettext(20, 12, 36, 21,buffer);
   clrscr();
   /* put selected characters back to the screen */
   gotoxy(20, 12);
   puttext(20, 12, 36, 21, buffer);
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.22 _setcursortype

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void _setcursortype(int cur_t);
```

Description

Selects cursor appearance.

Sets the cursor type to

_NOCURSOR	Turns off the cursor
_NORMALCURSOR	Normal underscore cursor
_SOLIDCURSOR	Solid block cursor

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main( )
{
  // tell the user what to do
  clrscr();
  cputs("Press any key three times.\n\r");
  cputs("Each time the cursor will change shape.\n\r");
  gotoxy(1,5);           // show a solid cursor
  cputs("Now the cursor is solid.\n\r");
  _setcursortype(_SOLIDCURSOR);
  while(!kbhit()) {};         // wait to proceed
  getch();
  gotoxy(1,5);           // remove the cursor
  cputs("Now the cursor is gone.");
  clreol();
  gotoxy(1,6);
  _setcursortype(_NOCURSOR);
  while(!kbhit()) {};         // wait to proceed
  getch();
  gotoxy(1,5);                 // show a normal cursor
  cputs("Now the cursor is normal.");
  clreol();
  gotoxy(1,6);
  _setcursortype(_NORMALCURSOR);
```

```
  while(!kbhit()) {};         // wait to proceed
  getch();
  clrscr();
  return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.23 textattr

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textattr(int newattr);
```

Description

Sets text attributes.

Note: Do not use this function in Win32 GUI applications.

textattr lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with textcolor and textbackground.)

This function does not affect any characters currently onscreen; it affects only those characters displayed by functions (such as cprintf) performing text mode, direct video output after this function is called.

The color information is encoded in the newattr parameter as follows:

In this 8-bit newattr parameter:

- bits 0-3 contain the 4-bit foreground color (0 to 15).
- bits 4-6 contain the 3-bit background color (0 to 7).
- bit 8 is the blink-enable bit.

If the blink-enable bit is on, the character blinks. This can be accomplished by adding the constant BLINK to the attribute.

- If you use the symbolic color constants defined in conio.h for creating text attributes with textattr, note the following limitations on the color you select for the background:
- You can select only one of the first eight colors for the background.
- You must shift the selected background color left by 4 bits to move it into the correct bit positions.

These symbolic constants are listed in the following table:

Return Value

None.

Example

```
#include <conio.h>
```

```
int main(void)
{
   int i;
   clrscr();
   for (i=0; i<9; i++)
   {
       textattr(i + ((i+1) << 4));
       cprintf("This is a test\r\n");
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.24 textbackground

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textbackground(int newcolor);
```

Description

Selects new text background color.

Note: Do not use this function in Win32 GUI applications.

textbackground selects the background color. This function works for functions that produce output in text mode directly to the screen. newcolor selects the new background color. You can set newcolor to an integer from 0 to 7, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called textbackground, all subsequent functions using direct video output (such as cprintf) will use newcolor. textbackground does not affect any characters currently onscreen.

The following table lists the symbolic constants and the numeric values of the allowable colors:

BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   int i, j;
   clrscr();
   for (i=0; i<9; i++)
   {
       for (j=0; j<80; j++)
          cprintf("C");
       cprintf("\r\n");
       textcolor(i+1);
       textbackground(i);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.25 textcolor

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textcolor(int newcolor);
```

Description

Selects new character color in text mode.

Note: Do not use this function in Win32 GUI applications.

textcolor selects the foreground character color. This function works for the console output functions. newcolor selects the new foreground color. You can set newcolor to an integer as given in the table below, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called textcolor, all subsequent functions using direct video output (such as cprintf) will use newcolor. textcolor does not affect any characters currently onscreen.

The following table lists the allowable colors (as symbolic constants) and their numeric values:

BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4

MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

You can make the characters blink by adding 128 to the foreground color. The predefined constant BLINK exists for this purpose.

For example:

```
textcolor(CYAN + BLINK);
```

Note: Some monitors do not recognize the intensity signal used to create the eight “light” colors (8-15). On such monitors, the light colors are displayed as their “dark” equivalents (0-7). Also, systems that do not display in color can treat these numbers as shades of one color, special patterns, or special attributes (such as underlined, bold, italics, and so on). Exactly what you’ll see on such systems depends on your hardware.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.26 textmode

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textmode(int newmode);
```

Description

Puts screen in text mode.

Note: Do not use this function in Win32 GUI applications.

textmode selects a specific text mode.

You can give the text mode (the argument newmode) by using a symbolic constant from the enumeration type text_modes (defined in conio.h).

The most commonly used text_modes type constants and the modes they specify are given in the following table. Some additional values are defined in conio.h.

LASTMODE	Previous text mode
BW40	Black and white, 40 columns
C40	Color, 40 columns
BW80	Black and white, 80 columns
C80	Color, 80 columns
MONO	Monochrome, 80 columns
C4350	EGA 43-line and VGA 50-line modes

When textmode is called, the current window is reset to the entire screen, and the current text attributes are reset to normal, corresponding to a call to normvideo.

Specifying LASTMODE to textmode causes the most recently selected text mode to be reselected.

textmode should be used only when the screen or window is in text mode (presumably to change to a different text mode). This is the only context in which textmode should be used. When the screen is in graphics mode, use restorecrtmode instead to escape temporarily to text mode.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   textmode(BW40);
   cprintf("ABC");
   getch();
   textmode(C40);
   cprintf("ABC");
   getch();
   textmode(BW80);
   cprintf("ABC");
   getch();
   textmode(C80);
   cprintf("ABC");
   getch();
   textmode(MONO);
   cprintf("ABC");
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.27 ungetch

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int ungetch(int ch);
```

Description

Pushes a character back to the keyboard buffer.

Note: Do not use this function in Win32 GUI applications.

ungetch pushes the character ch back to the console, causing ch to be the next character read. The ungetch function fails if it is called more than once before the next read.

Return Value

On success, ungetch returns the character ch.

On error, it returns EOF.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
int main( void )
{
   int i=0;
   char ch;
   puts("Input an integer followed by a char:");
   /* read chars until non digit or EOF */
   while((ch = getche()) != EOF && isdigit(ch))
      i = 10 * i + ch - 48; /* convert ASCII into int value */
   /* if non digit char was read, push it back into input buffer */
   if (ch != EOF)
      ungetch(ch);
   printf("\n\ni = %d, next char in buffer = %c\n", i, getch());
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.28 wherex

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int wherex(void);
```

Description

Gives horizontal cursor position within window.

Note: Do not use this function in Win32 GUI applications.

wherex returns the x-coordinate of the current cursor position (within the current text window).

Return Value

wherex returns an integer in the range 1 to the number of columns in the current video mode.

Example

```
#include <conio.h>
int main(void)
{
   clrscr();
   gotoxy(10,10);
   cprintf("Current location is X: %d  Y: %d\r\n", wherex(), wherey());
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.29 wherey

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int wherey(void);
```

Description

Gives vertical cursor position within window.

Note: Do not use this function in Win32 GUI applications.

wherey returns the y-coordinate of the current cursor position (within the current text window).

Return Value

wherey returns an integer in the range 1 to the number of rows in the current video mode.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.3.30 window

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void window(int left, int top, int right, int bottom);
```

Description

Defines active text mode window.

Note: Do not use this function in Win32 GUI applications.

window defines a text window onscreen. If the coordinates are in any way invalid, the call to window is ignored.

left and top are the screen coordinates of the upper left corner of the window.

right and bottom are the screen coordinates of the lower right corner.

The minimum size of the text window is one column by one line. The default window is full screen, with the coordinates:

1,1,C,R

where C is the number of columns in the current video mode, and R is the number of rows.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   window(10,10,40,11);
   textcolor(BLACK);
   textbackground(WHITE);
   cprintf("This is a test\r\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.4 ctype.h

The following functions, macros, and classes are provided in `ctype.h`:

1.1.2.4.1 isalnum, __iscsym, iswalnum, _ismbcalnum

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isalnum(int c);

int __iscsym(int c);

int iswalnum(wint_t c);

int _ismbcalnum(unsigned int c);
```

Description

Tests for an alphanumeric character.

isalnum is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a letter (A to Z or a to z) or a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

It is a predicate returning nonzero for true and 0 for false.

isalnum returns nonzero if c is a letter or a digit.

__iscsym returns nonzero if c is a letter, underscore, or digit.

iswalnum returns nonzero if iswalpha or iswdigit return true for c.

_ismbcalnum returns true if and only if the argument c is a single-byte ASCII English letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';

  if (isalnum(c))
    printf("%c is alphanumeric\n",c);
  else
    printf("%c is not alphanumeric\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isalnum	+	+	+	+
__iscsym		+		
_ismbcalnum		+		
iswalnum		+	+	+

1.1.2.4.2 isalpha, __iscsymf, iswalpha, _ismbcalpha

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isalpha(int c);

int __iscsymf(int c);

int iswalpha(wint_t c);

int _ismbcalpha(unsigned int c);
```

Description

Classifies an alphabetical character.

isalpha is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a letter (A to Z or a to z).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isalpha returns nonzero if c is a letter.

__iscsymf returns true if and only if the argument c is a letter or an underscore.

iswalpha returns nonzero if c is a **wchar_t** in the character set defined by the implementation.

_ismbcalpha returns true if and only if the argument c is a single-byte ASCII English letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';

  if (isalpha(c))
    printf("%c is alphabetical\n",c);
  else
    printf("%c is not alphabetical\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isalpha	+	+	+	+
__iscsymf		+		
_ismbcalpha		+		
iswalpha		+	+	+

1.1.2.4.3 isascii, iswascii

Header File

ctype.h, wctype.h

Category

Classification Routines

Prototype

```
int isascii(int c);
int iswascii(wint_t c);
```

Description

Character classification macro.

These functions depend on the LC_CTYPE

isascii is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false.

isascii is defined on all integer values.

Return Value

isascii returns nonzero if c is in the range 0 to 127 (0x00-0x7F).

iswascii returns nonzero if c is is a wide-character representation of an ASCII character.

Each of these routines returns 0 if c does not satisfy the test condition.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <stdio.h>
int main(void)
{
  char c = 'C';
  if (isascii(c))
    printf("%c is ascii\n",c);
  else
    printf("%c is not ascii\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isascii		+		
iswascii		+		

1.1.2.4.4 iscntrl, iswcntrl

Header File

ctype.h

Category

Classification Routines

Prototype

```
int iscntrl(int c);

int iswcntrl(wint_t c);
```

Description

Tests for a control character.

iscntrl is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a delete character or control character (0x7F or 0x00 to 0x1F).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

iscntrl returns nonzero if c is a delete character or ordinary control character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (iscntrl(c))
    printf("%c is a control character\n",c);
  else
    printf("%c is not a control character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
iscntrl	+	+	+	+
iswcntrl		+	+	+

1.1.2.4.5 isdigit, iswdigit, _ismbcdigit

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isdigit(int c);

int iswdigit(wint_t c);

int _ismbcdigit(unsigned int c);
```

Description

Tests for decimal-digit character.

isdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isdigit returns nonzero if c is a digit.

_ismbcdigit returns true if and only if the argument c is a single-byte representation of an ASCII digit.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isdigit(c))
    printf("%c is a digit\n",c);
  else
    printf("%c is not a digit\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isdigit	+	+	+	+
_ismbcdigit		+		
iswdigit		+	+	+

1.1.2.4.6 isgraph, iswgraph, _ismbcgraph

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isgraph(int c);
```

```
int iswgraph(wint_t c);
```

```
int _ismbcgraph( unsigned int c);
```

Description

Tests for printing character.

isgraph is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a printing character except blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isgraph returns nonzero if c is a printing character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isgraph(c))
    printf("%c is a graphic character\n",c);
  else
    printf("%c is not a graphic character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isgraph	+	+	+	+
_ismbcgraph		+		
iswgraph		+	+	+

1.1.2.4.7 islower, iswlower, _ismbclower

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int islower(int c);
```

```
int iswlower(wint_t c);
```

```
int _ismbclower(unsigned int c);
```

Description

Tests for lowercase character.

islower is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a lowercase letter (a to z).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

islower returns nonzero if c is a lowercase letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (islower(c))
    printf("%c is a lowercase character\n",c);
  else
    printf("%c is not a lowercase character\n",c);
```

```
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
islower	+	+	+	+
_ismbclower		+		
iswlower		+	+	+

1.1.2.4.8 isprint, iswprint, _ismbcprint

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isprint(int c);
```

```
int iswprint(wint_t c);
```

```
int _ismbcprint(unsigned int c);
```

Description

Tests for printing character.

isprint is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a printing character including the blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isprint returns nonzero if c is a printing character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isprint(c))
    printf("%c is a printable character\n",c);
  else
    printf("%c is not a printable character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isprint	+	+	+	+
_ismbcprint		+		
iswprint		+		

1.1.2.4.9 ispunct, iswpunct, _ismbcpunct

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int ispunct(int c);

int iswpunct(wint_t c);

int _ismbcpunct(unsigned int c);
```

Description

Tests for punctuation character.

ispunct is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is any printing character that is neither an alphanumeric nor a blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

ispunct returns nonzero if c is a punctuation character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (ispunct(c))
    printf("%c is a punctuation character\n",c);
  else
    printf("%c is not a punctuation character\n",c);

  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ispunct	+	+	+	+
_ismbcpunt		+		
iswpunct		+	+	+

1.1.2.4.10 isspace, iswspace, _ismbcspace

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isspace(int c);

int iswspace(wint_t c);

int _ismbcspace(unsigned int c);
```

Description

Tests for space character.

isspace is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isspace returns nonzero if c is a space, tab, carriage return, new line, vertical tab, formfeed (0x09 to 0x0D, 0x20), or any other locale-defined space character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isspace(c))
    printf("%c is white space\n",c);
  else
    printf("%c is not white space\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isspace	+	+	+	+
_ismbcspace		+		
iswspace		+	+	+

1.1.2.4.11 isupper, iswupper, _ismbcupper

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isupper(int c);

int iswupper(wint_t c);

int _ismbcupper(unsigned int c);
```

Description

Tests for uppercase character.

isupper is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is an uppercase letter (A to Z).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isupper returns nonzero if c is an uppercase letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isupper(c))
    printf("%c is an uppercase character\n",c);
  else
    printf("%c is not an uppercase character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isupper	+	+	+	+
_ismbcupper		+		
iswupper		+	+	+

1.1.2.4.12 isxdigit, iswxdigit

Header File

ctype.h, wctype.h

Category

Classification Routines

Prototype

```
int isxdigit(int c);
```

```
int iswxdigit(wint_t c);
```

Description

Tests for hexadecimal character.

isxdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (#undef) it.

Return Value

isxdigit returns nonzero if c is a hexadecimal digit (0 to 9, A to F, a to f) or any other hexadecimal digit defined by the locale.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isxdigit(c))
    printf("%c is a hexadecimal digit\n",c);
  else
    printf("%c is not a hexadecimal digit\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isxdigit	+	+	+	+
iswxdigit		+	+	+

1.1.2.4.13 toascii

Header File

ctype.h

Category

Conversion Routines

Prototype

```
int toascii(int c);
```

Description

Translates characters to ASCII format.

toascii is a macro that converts the integer c to ASCII by clearing all but the lower 7 bits; this gives a value in the range 0 to 127.

Return Value

toascii returns the converted value of c.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.4.14 _tolower

Header File

ctype.h

Category

Conversion Routines

Prototype

```
int _tolower(int ch);
```

Description

_tolower is a macro that does the same conversion as tolower, except that it should be used only when ch is known to be uppercase (AZ).

To use _tolower, you must include ctype.h.

Return Value

_tolower returns the converted value of ch if it is uppercase; otherwise, the result is undefined.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "THIS IS A STRING.";
   length = strlen(string);
   for (i = 0; i < length; i++) {
       if ((string[i] >= 'A') && (string[i] <= 'Z')){
          string[i] = _tolower(string[i]);
       }
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.4.15 tolower, _mbctolower, towlower

Header File

ctype.h, mbstring.h

Category

Conversion Routines

Prototype

```
int tolower(int ch);

int towlower(wint_t ch); // Unicode version

unsigned int _mbctolower(unsigned int c);
```

Description

Translates characters to lowercase.

tolower is a function that converts an integer ch (in the range EOF to 255) to its lowercase value (a to z; if it was uppercase, A to Z). All others are left unchanged.

Return Value

tolower returns the converted value of ch if it is uppercase; it returns all others unchanged.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "THIS IS A STRING";
   length = strlen(string);
   for (i=0; i<length; i++)
   {
       string[i] = tolower(string[i]);
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tolower	+	+	+	+
_mbctolower		+		
towlower		+	+	+

1.1.2.4.16 _toupper

Header File

ctype.h

Category

Conversion Routines

Prototype

```
int _toupper(int ch);
```

Description

Translates characters to uppercase.

_toupper is a macro that does the same conversion as toupper, except that it should be used only when ch is known to be lowercase (a to z).

To use _toupper, you must include ctype.h.

Return Value

_toupper returns the converted value of ch if it is lowercase; otherwise, the result is undefined.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "this is a string.";
```

```
   length = strlen(string);
   for (i = 0; i < length; i++) {
      if ((string[i] >= 'a') && (string[i] <= 'z')){
         string[i] = _toupper(string[i]);
         }
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.4.17 toupper, _mbctoupper, towupper

Header File

ctype.h, mbstring.h

Category

Conversion Routines

Prototype

```
int toupper(int ch);

int towupper(wint_t ch); // Unicode version

unsigned int _mbctoupper(unsigned int c);
```

Description

Translates characters to uppercase.

toupper is a function that converts an integer ch (in the range EOF to 255) to its uppercase value (A to Z; if it was lowercase, a to z). All others are left unchanged.

towupper is the Unicode version of toupper. It is available when Unicode is defined.

Return Value

toupper returns the converted value of ch if it is lowercase; it returns all others unchanged.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "this is a string";
   length = strlen(string);
   for (i=0; i<length; i++)
   {
      string[i] = toupper(string[i]);
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
toupper	+	+	+	+
_mbctoupper		+		
towupper		+	+	+

1.1.2.4.18 _ctype

Header File

ctype.h

Syntax

```
extern char _ctype[];
```

Description

_ctype is an array of character attribute information indexed by ASCII value + 1. Each entry is a set of bits describing the character. This array is used by isdigit, isprint, and so on.

1.1.2.4.19 _IS_xxx #defines

Header File

ctype.h

Description

Bit settings in the _ctype[] used by the is... character macros.

Name	Meaning
_IS_SP	Is space
_IS_DIG	Is digit
_IS_UPP	Is uppercase
_IS_LOW	Is lowercase
_IS_HEX	[A-F] or [a-f]
_IS_CTL	Control
_IS_PUN	Punctuation

1.1.2.5 delayimp.h

The following functions, macros, and classes are provided in `delayimp.h`:

1.1.2.5.1 __FUnloadDelayLoadedDLL

Header File

delayimp.h

Prototype

```
BOOL WINAPI __FUnloadDelayLoadedDLL(LPCSTR szDll);
```

Description

Unloads a delay loaded DLL.

szDll is pointer to a name to unload, or NULL to unload all the delay load DLLs in the list.

Return Value

On successful completion __FUnloadDelayLoadedDLL returns **true**.

On error it returns **false**.

1.1.2.5.2 __pfnDliNotifyHook, __pfnDliFailureHook

Header File

delayimp.h

Category

Delay load hook notification Routines

Prototype

```
typedef FARPROC (WINAPI *DelayedLoadHook)(dliNotification dliNotify,
DelayLoadInfo * pdli);
```

Description

The delay load mechanism provides two hooks for you to modify the runtime behavior of a delay loaded DLL. By writing your own hook functions using the function signature below and assigning this to the two hooks you can modify the delay load process.

DelayLoadProc structure

```
typedef struct DelayLoadProc
{
BOOL fImportByName;
union
{
LPCSTR szProcName;
DWORD dwOrdinal;
};
} DelayLoadProc;
```

ImgDelayDescr structure

```
typedef struct ImgDelayDescr
{
DWORD grAttrs; /* attributes */
LPCSTR szName; /* pointer to dll name */
```

```
HMODULE hmod; /* address of module handle */
IMAGE_THUNK_DATA * pIAT; /* address of the IAT */
IMAGE_THUNK_DATA * pINT; /* address of the INT */
IMAGE_THUNK_DATA * pBoundIAT; /* address of the optional bound IAT */
IMAGE_THUNK_DATA * pUnloadIAT; /* address of optional copy of
original IAT */
DWORD dwTimeStamp; /* 0 if not bound, */
/* O.W. date/time stamp of DLL bound
to (Old BIND) */
} ImgDelayDescr;
```

DelayLoadInfo structure

```
typedef struct DelayLoadInfo
{
DWORD cb; /* size of structure */
const ImgDelayDescr * pidd; /* raw form of data (everything is
there) */
FARPROC * ppfn; /* points to address of function to
load */
LPCSTR szDll; /* name of dll */
DelayLoadProc dlp; /* name or ordinal of procedure */
HMODULE hmodCur; /* the hInstance of the library we
have loaded */
FARPROC pfnCur; /* the actual function that will be
called */
DWORD dwLastError;/* error received (if an error
notification) */
} DelayLoadInfo, *PDelayLoadInfo;
```

Delay load import hook notifications

The following structure is the enumerations that are defined for the hook notification events:

```
typedef enum
{
dliNoteStartProcessing, /* used to bypass or note helper only */
dliNotePreLoadLibrary, /* called just before LoadLibrary, can */
/* override w/ new HMODULE return val */
dliNotePreGetProcAddress, /* called just before GetProcAddress, can */
/* override w/ new FARPROC return value */
dliFailLoadLibrary, /* failed to load library, fix it by */
```

```
/* returning a valid HMODULE */
dliFailGetProcAddress, /* failed to get proc address, fix it by */
/* returning a valid FARPROC */
dliNoteEndProcessing, /* called after all processing is done, */
/* no bypass possible at this point */
/* except by longjmp(), throw(), or
RaiseException. */
} dliNotification;
```

Hook pointers

The “notify hook” gets called for every call to the delay load helper. This allows a user to hook every call and skip the delay load helper entirely.

```
extern DelayedLoadHook _EXPDATA __pfnDliNotifyHook;
dliNotify ==
{
dliNoteStartProcessing |
dliNotePreLoadLibrary |
dliNotePreGetProcAddress |
dliNoteEndProcessing
}
```

Note: The “failure” hook is assigned to:

```
extern DelayedLoadHook _EXPDATA __pfnDliFailureHook;
```

This hook is called with the following notification flags:

```
dliNotify ==
{
dliFailLoadLibrary |
dliFailGetProcAddress
}
```

For further information on when this notify eventsd occur during the delay load process, please see delayhlp.c

1.1.2.6 dir.h

The following functions, macros, and classes are provided in `dir.h`:

1.1.2.6.1 chdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int chdir(const char *path);
```

```
int _wchdir(const wchar_t *path);
```

Description

Changes current directory.

chdir causes the directory specified by path to become the current working directory; path must specify an existing directory.

A drive can also be specified in the path argument, such as

```
chdir("a:\\BC")
```

but this method changes only the current directory on that drive; it does not change the active drive.

- Under Windows, only the current process is affected.

Return Value

Upon successful completion, chdir returns a value of 0. Otherwise, it returns a value of -1, and the global variable errno is set to

ENOENT	Path or file name not found

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
char old_dir[MAXDIR];
char new_dir[MAXDIR];
int main(void)
{
   if (getcurdir(0, old_dir))
   {
      perror("getcurdir()");
      exit(1);
   }
   printf("Current directory is: \\%s\n", old_dir);
   if (chdir("\\"))
   {
      perror("chdir()");
      exit(1);
   }
   if (getcurdir(0, new_dir))
   {
      perror("getcurdir()");
      exit(1);
   }
   printf("Current directory is now: \\%s\n", new_dir);
   printf("\nChanging back to original directory: \\%s\n", old_dir);
   if (chdir(old_dir))
   {
      perror("chdir()");
      exit(1);
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
chdir	+	+		
_wchdir		NT only		

1.1.2.6.2 findclose, _wfindclose

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int findclose(struct ffblk *ffblk );
int _wfindclose(struct _wffblk *ffblk );
```

Description

findclose closes any handles and frees up any dynamic memory associated with previous calls to findfirst and findnext.

Return Value

findclose returns 0 on successfully closing the handle. On failure:

- -1 is returned
- errno is set to

EINVDAT	Invalid data

Portability

	POSIX	Win32	ANSI C	ANSI C++
findclose		+		
_wfindclose		NT only		

See Also

findfirst (see page 262)

findnext (see page 265)

1.1.2.6.3 findfirst, _wfindfirst

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
int _wfindfirst( const wchar_t *pathname, struct _wffblk *ffblk, int attrib);
```

Description

Searches a disk directory.

findfirst begins a search of a disk directory for files specified by attributes or wildcards.

pathname is a string with an optional drive specifier path and file name of the file to be found. Only the file name portion can contain wildcard match characters (such as ? or *). If a matching file is found the ffblk structure is filled with the file-directory information.

When Unicode is defined, the _wfindfirst function uses the following _wffblk structure.

```
struct _wffblk {
long ff_reserved;
long ff_fsize;
unsigned long ff_attrib;
unsigned short ff_ftime;
unsigned short ff_fdate;
wchar_t ff_name[256];
};
```

For Win32, the format of the structure ffblk is as follows:

```
struct ffblk {
long ff_reserved;
long ff_fsize; /* file size */
unsigned long ff_attrib; /* attribute found */
unsigned short ff_ftime; /* file time */
unsigned short ff_fdate; /* file date */
char ff_name[256]; /* found file name */
};
```

attrib is a file-attribute byte used in selecting eligible files for the search. attrib should be selected from the following constants defined in dos.h:

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

A combination of constants can be OR’ed together.

For more detailed information about these attributes refer to your operating system documentation.

ff_ftime and ff_fdate contain bit fields for referring to the current date and time. The structure of these fields was established by

the operating system. Both are 16-bit structures divided into three fields.

ff_ftime:

Bits 0 to 4	The result of seconds divided by 2 (for example 10 here means 20 seconds)
Bits 5 to 10	Minutes
Bits 11 to 15	Hours

ff_fdate:

Bits 0-4	Day
Bits 5-8	Month
Bits 9-15	Years since 1980 (for example 9 here means 1989)

The structure ftime declared in io.h uses time and date bit fields similar in structure to ff_ftime and ff_fdate.

Return Value

findfirst returns 0 on successfully finding a file matching the search pathname.

When no more files can be found, or if there is an error in the file name:

- -1 is returned
- errno is set to

ENOENT	Path or file name not found

- _doserrno is set to one of the following values:

ENMFILE	No more files
ENOENT	Path or file name not found

Example

```
/* findfirst and findnext example */
#include <stdio.h>
#include <dir.h>
int main(void)
{
   struct ffblk ffblk;
   int done;
   printf("Directory listing of *.*\n");
   done = findfirst("*.*",&ffblk,0);
   while (!done)
   {
      printf("  %s\n", ffblk.ff_name);
      done = findnext(&ffblk);
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
findfirst		+		

_wfindfirst		NT only		

See Also

findclose (see page 262)

findnext (see page 265)

1.1.2.6.4 findnext, _wfindnext

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int findnext(struct ffblk *ffblk );
int _wfindnext(struct _wffblk *ffblk );
```

Description

Continues findfirst search.

findnext is used to fetch subsequent files that match the pathname given in findfirst. ffblk is the same block filled in by the findfirst call. This block contains necessary information for continuing the search. One file name for each call to findnext will be returned until no more files are found in the directory matching the pathname.

Return Value

findnext returns 0 on successfully finding a file matching the search pathname. When no more files can be found or if there is an error in the file name

-1 is returned

errno is set to

ENOENT	Path or file name not found

_doserrno is set to one of the following values:

ENMFILE	No more files
ENOENT	Path or file name not found

Portability

	POSIX	Win32	ANSI C	ANSI C++
findnext		+		
_wfindnext		NT only		

See Also

findfirst (see page 262)

findnext

1.1.2.6.5 fnmerge, _wfnmerge

Header File

dir.h

Category

Directory Control Routines

Prototype

```
void fnmerge(char *path, const char *drive, const char *dir, const char *name, const char *ext);
```

```
void _wfnmerge(wchar_t *path, const wchar_t *drive, const wchar_t *dir, const wchar_t *name, const wchar_t *ext );
```

Description

Builds a path from component parts.

fnmerge makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where:

drive	=	X
dir	=	\\DIR\\SUBDIR\\
name	=	NAME
ext	=	.EXT

If drive is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If dir is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If name is empty or NULL, no file name is inserted in the path name.

If ext is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

fnmerge assumes there is enough space in path for the constructed path name. The maximum constructed length is MAXPATH. MAXPATH is defined in dir.h.

fnmerge and fnsplit are invertible; if you split a given path with fnsplit then merge the resultant components with fnmerge you end up with path.

Return Value

None.

Example

```
#include <string.h>
#include <stdio.h>
#include <dir.h>
int main(void)
{
```

```
    char s[MAXPATH];
    char drive[MAXDRIVE];
    char dir[MAXDIR];
    char file[MAXFILE];
    char ext[MAXEXT];
    getcwd(s,MAXPATH);              /* get the current working directory */
    strcat(s,"\\");                 /* append on a trailing character */
    fnsplit(s,drive,dir,file,ext);  /* split the string to separate elems */
    strcpy(file,"DATA");
    strcpy(ext,".TXT");
    fnmerge(s,drive,dir,file,ext);   /* merge everything into one string */
    puts(s);                         /* display resulting string */
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fnmerge		+		
_wfnmerge		NT only		

1.1.2.6.6 fnsplit, _wfnsplit

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext);
```

```
int _wfnsplit(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t *name, wchar_t *ext );
```

Description

Splits a full path name into its components.

fnsplit takes a file's full path name (path) as a string in the form X:\DIR\SUBDIR\NAME.EXT and splits path into its four components. It then stores those components in the strings pointed to by drive, dir, name, and ext. All five components must be passed but any of them can be a null which means the corresponding component will be parsed but not stored. If any path component is null, that component corresponds to a non-NULL, empty string.

The maximum sizes for these strings are given by the constants MAXDRIVE, MAXDIR, MAXPATH, MAXFILE, and MAXEXT (defined in dir.h) and each size includes space for the null-terminator.

fnsplit assumes that there is enough space to store each non-null component.

- When fnsplit splits path it treats the punctuation as follows:
- drive includes the colon (C:, A:, and so on)
- dir includes the leading and trailing backslashes (\BC\include\, \source\ ,and so on)
- name includes the file name
- ext includes the dot preceding the extension (.C, .EXE, and so on).

fnmerge and fnsplit are invertible; if you split a given path with fnsplit then merge the resultant components with fnmerge you end up with path.

Return Value

fnsplit returns an integer (composed of five flags defined in dir.h) indicating which of the full path name components were present in path. These flags and the components they represent are

EXTENSION	An extension
FILENAME	A file name
DIRECTORY	A directory (and possibly subdirectories)
DRIVE	A drive specification (see dir.h)
WILDCARDS	Wildcards (* or ?)

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <dir.h>
int main(void)
{
    char *s;
    char drive[MAXDRIVE];
    char dir[MAXDIR];
    char file[MAXFILE];
    char ext[MAXEXT];
    int flags;
    s=getenv("COMSPEC"); /* get the comspec environment parameter */
    flags=fnsplit(s,drive,dir,file,ext);
    printf("Command processor info:\n");
    if(flags & DRIVE)
       printf("\tdrive: %s\n",drive);
    if(flags & DIRECTORY)
       printf("\tdirectory: %s\n",dir);
    if(flags & FILENAME)
       printf("\tfile: %s\n",file);
    if(flags & EXTENSION)
       printf("\textension: %s\n",ext);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fnsplit		+		
_wfnsplit		NT only		

1.1.2.6.7 getcurdir, _wgetcurdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int getcurdir(int drive, char *directory);
```

```
int _wgetcurdir(int drive, wchar_t *directory );
```

Description

Gets current directory for specified drive.

getcurdir gets the name of the current working directory for the drive indicated by drive. drive specifies a drive number (0 for default, 1 for A, and so on). directory points to an area of memory of length MAXDIR where the null-terminated directory name will be placed. The name does not contain the drive specification and does not begin with a backslash.

Return Value

getcurdir returns 0 on success or -1 in the event of error.

Example

```
#include <dir.h>
#include <stdio.h>
#include <string.h>
char *current_directory(char *path)
{
  strcpy(path, "X:\\");      /* fill string with form of response: X:\ */
  path[0] = 'A' + getdisk();    /* replace X with current drive letter */
  getcurdir(0, path+3);  /* fill rest of string with current directory */
  return(path);
}
int main(void)
{
  char curdir[MAXPATH];
  current_directory(curdir);
  printf("The current directory is %s\n", curdir);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getcurdir		+		
_wgetcurdir		NT only		

1.1.2.6.8 getcwd, _wgetcwd

Header File

dir.h

Category

Directory Control Routines

Prototype

```
char *getcwd(char *buf, int buflen);
```

```
wchar_t *_wgetcwd(wchar_t *buf, int buflen);
```

Description

Gets current working directory.

getcwd gets the full path name (including the drive) of the current working directory, up to buflen bytes long and stores it in buf. If the full path name length (including the null terminator) is longer than buflen bytes, an error occurs.

If buf is NULL, a buffer buflen bytes long is allocated for you with malloc. You can later free the allocated buffer by passing the return value of getcwd to the function free.

Return Value

- getcwd returns the following values:
- If buf is not NULL on input, getcwd returns buf on success, NULL on error.
- If buf is NULL on input, getcwd returns a pointer to the allocated buffer.

In the event of an error return, the global variable errno is set to one of the following values:

ENODEV	No such device
ENOMEM	Not enough memory to allocate a buffer (buf is NULL)
ERANGE	Directory name longer than buflen (buf is not NULL)

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
   char buffer[MAXPATH];
   getcwd(buffer, MAXPATH);
   printf("The current directory is: %s\n", buffer);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getcwd	+	+		
_wgetcwd		NT only		

1.1.2.6.9 getdisk, setdisk

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int getdisk(void);
```

```
int setdisk(int drive);
```

Description

Gets or sets the current drive number.

getdisk gets the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on.

setdisk sets the current drive to the one associated with drive: 0 for A, 1 for B, 2 for C, and so on.

The setdisk function changes the current drive of the parent process.

Return Value

getdisk returns the current drive number. setdisk returns the total number of drives available.

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
  int disk, maxdrives = setdisk(2);
  disk = getdisk() + 'A';
  printf("\nThe number of logical drives is:%d\n", maxdrives);
  printf("The current drive is: %c\n", disk);
  return 0;
 }
```

1.1.2.6.10 mkdir, _wmkdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int mkdir(const char *path);
```

```
int _wmkdir(const wchar_t *path);
```

Description

Creates a directory.

mkdir is available on UNIX, though it then takes an additional parameter.

mkdir creates a new directory from the given path name path.

Return Value

mkdir returns the value 0 if the new directory was created.

A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	No such file or directory

Example

```
#include <stdio.h>
#include <process.h>
#include <dir.h>
#define DIRNAME "testdir.$$$"
int main(void)
{
    int stat;
    stat = mkdir(DIRNAME);
    if (!stat)
           printf("Directory created\n");
    else
    {
       printf("Unable to create directory\n");
       exit(1);
    }
    getchar();
    system("dir/p");
    getchar();
    stat = rmdir(DIRNAME);
```

```
    if (!stat)
          printf("\nDirectory deleted\n");
    else
    {
          perror("\nUnable to delete directory\n");
       exit(1);
    }
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
mkdir	+	+		
_wmkdir		NT only		

1.1.2.6.11 _mktemp, _wmktemp

Header File

dir.h

Category

Directory Control Routines

Prototype

```
char *_mktemp(char *template);
```

```
wchar_t *_wmktemp(wchar_t *template);
```

Description

Makes a unique file name.

_mktemp replaces the string pointed to by template with a unique file name and returns template.

template should be a null-terminated string with six trailing Xs. These Xs are replaced with a unique collection of letters plus a period, so that there are two letters, a period, and three suffix letters in the new file name.

Starting with AA.AAA, the new file name is assigned by looking up the name on the disk and avoiding pre-existing names of the same format.

Return Value

If a unique name can be created and template is well formed, _mktemp returns the address of the template string. Otherwise, it returns null.

Example

```
#include <dir.h>
#include <stdio.h>
int main(void)
{
    /* fname defines the template for the
      temporary file.  */
    char *fname = "TXXXXXX", *ptr;
    ptr = mktemp(fname);
    printf("%s\n",ptr);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
mktemp	+	+		
_wmktemp		+		

1.1.2.6.12 _rmdir, _wrmdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int _rmdir(const char *path);
```

```
int _wrmdir(const wchar_t *path);
```

Description

Removes a directory.

_rmdir deletes the directory whose path is given by path. The directory named by path

- must be empty
- must not be the current working directory
- must not be the root directory

Return Value

_rmdir returns 0 if the directory is successfully deleted. A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file function not found

Example

```
#include <stdio.h>
#include <process.h>
#include <dir.h>
#define DIRNAME "testdir.$$$"
int main(void)
{
    int stat;
    stat = mkdir(DIRNAME);
    if (!stat)
           printf("Directory created\n");
    else
    {
       printf("Unable to create directory\n");
       exit(1);
    }
    getchar();
    system("dir/p");
    getchar();
```

```
    stat = rmdir(DIRNAME);
    if (!stat)
           printf("\nDirectory deleted\n");
    else
    {
           perror("\nUnable to delete directory\n");
       exit(1);
    }
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rmdir	+	+		
_wrmdir		NT only		

1.1.2.6.13 searchpath, wsearchpath

Header File

dir.h

Category

Miscellaneous Routines

Prototype

```
char *searchpath(const char *file);
```

```
wchar_t *wsearchpath( const wchar_t *file );
```

Description

Searches the operating system path for a file.

searchpath attempts to locate file, searching along the operating system path, which is the PATH=... string in the environment. A pointer to the complete path-name string is returned as the function value.

searchpath searches for the file in the current directory of the current drive first. If the file is not found there, the PATH environment variable is fetched, and each directory in the path is searched in turn until the file is found, or the path is exhausted.

When the file is located, a string is returned containing the full path name. This string can be used in a call to access the file (for example, with fopen or exec...).

The string returned is located in a static buffer and is overwritten on each subsequent call to searchpath.

Return Value

searchpath returns a pointer to a file name string if the file is successfully located; otherwise, searchpath returns null.

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
    char *p;
    /* Looks for ILINK32 and returns a pointer
       to the path  */
    p = searchpath("ILINK32.EXE");
    printf("Search for ILINK32.EXE : %s\n", p);
```

```
    /* Looks for nonexistent file  */
    p = searchpath("NOTEXIST.FIL");
    printf("Search for NOTEXIST.FIL : %s\n", p);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
searchpath		+		
wsearchpath		NT only		

1.1.2.6.14 Bit Definitions for fnsplit

Header File

dir.h

Description

Bit definitions returned from fnsplit to identify which pieces of a file name were found during the split.

Flag	Component
DIRECTORY	Path includes a directory (and possibly subdirectories)
DRIVE	Path includes a drive specification (see DIR.H)
EXTENSION	Path includes an extension
FILENAME	Path includes a file name
WILDCARDS	Path contains wildcards (* or ?)

1.1.2.6.15 MAXxxxx #defines (fnsplit)

Header File

dir.h

Description

These symbols define the maximum number of characters in a file specification for fnsplit (including room for a terminating NULL).

Name	Meaning
MAXPATH	Complete file name with path
MAXDRIVE	Disk drive (e.g., "A:")
MAXDIR	File subdirectory specification
MAXFILE	File name without extension
MAXEXT	File extension

1.1.2.7 direct.h

The following functions, macros, and classes are provided in `direct.h`:

1.1.2.7.1 _chdrive

Header File

direct.h

Category

Directory Control Routines

Prototype

```
int _chdrive(int drive);
```

Description

Sets current disk drive.

_chdrive sets the current drive to the one associated with drive: 1 for A, 2 for B, 3 for C, and so on.

This function changes the current drive of the parent process.

Return Value

_chdrive returns 0 if the current drive was changed successfully; otherwise, it returns -1.

Example

```
 #include <stdio.h>
#include <direct.h>
int main(void)
{
  if (_chdrive(3) == 0)
    printf("Successfully changed to drive C:\n");
  else
    printf("Cannot change to drive C:\n");
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.7.2 _getdcwd, _wgetdcwd

Header File

direct.h

Category

Directory Control Routines

Prototype

```
char * _getdcwd(int drive, char *buffer, int buflen);
```

```
wchar_t * _wgetdcwd(int drive, wchar_t *buffer, int buflen);
```

Description

Gets current directory for specified drive.

_getdcwd gets the full path name of the working directory of the specified drive (including the drive name), up to buflen bytes long, and stores it in buffer. If the full path name length (including the null-terminator) is longer than buflen, an error occurs. The drive is 0 for the default drive, 1=A, 2=B, and so on.

If the working directory is the root directory, the terminating character for the full path is a backslash. If the working directory is a subdirectory, there is no terminating backslash after the subdirectory name.

If buffer is NULL, _getdcwd allocates a buffer at least buflen bytes long. You can later free the allocated buffer by passing the _getdcwd return value to the free function.

Return Value

If successful, _getdcwd returns a pointer to the buffer containing the current directory for the specified drive.

Otherwise it returns NULL, and sets the global variable errno to one of the following values:

ENOMEM	Not enough memory to allocate a buffer (buffer is NULL)
ERANGE	Directory name longer than buflen (buffer is not NULL)

Example

```
#include <direct.h>
#include <stdio.h>
char buf[65];
void main()
{
  if (_getdcwd(3, buf, sizeof(buf)) == NULL)
    perror("Unable to get current directory of drive C");
  else
    printf("Current directory of drive C is %s\n",buf);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_getdcwd		+		
_wgetdcwd		NT only		

1.1.2.8 dirent.h

The following functions, macros, and classes are provided in `dirent.h`:

1.1.2.8.1 closedir, wclosedir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
void closedir(DIR *dirp);
```

```
void wclosedir(wDIR *dirp);
```

Description

Closes a directory stream.

closedir is available on POSIX-compliant UNIX systems.

The closedir function closes the directory stream dirp, which must have been opened by a previous call to opendir. After the stream is closed, dirp no longer points to a valid directory stream.

wclosedir is the Unicode version of closedir.

Return Value

If closedir is successful, it returns 0. Otherwise, closedir returns -1 and sets the global variable errno to

EBADF	The dirp argument does not point to a valid open directory stream

Example

```
/* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
}

void main(int argc,char *argv[])
{
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.8.2 opendir, wopendir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
DIR *opendir(const char *dirname);
wDIR *wopendir(const wchar_t *dirname);
```

Description

Opens a directory stream for reading.

opendir is available on POSIX-compliant UNIX systems.

The opendir function opens a directory stream for reading. The name of the directory to read is dirname. The stream is set to read the first entry in the directory.

A directory stream is represented by the DIR structure, defined in dirent.h. This structure contains no user-accessible fields. Multiple directory streams can be opened and read simultaneously. Directory entries can be created or deleted while a directory stream is being read.

Use the readdir function to read successive entries from a directory stream. Use the closedir function to remove a directory stream when it is no longer needed.

Return Value

On success, opendir returns a pointer to a directory stream that can be used in calls to readdir, rewinddir, and closedir.

On error (If the directory cannot be opened), the functino returns NULL and sets the global variable errno to

ENOENT	The directory does not exist
ENOMEM	Not enough memory to allocate a DIR object

Example

```
 /* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
```

```
      perror("Unable to open directory");
      exit(1);
    }
    while ((ent = readdir(dir)) != NULL)
      printf("%s\n",ent->d_name);

    printf("Second pass on '%s':\n",dirname);
    rewinddir(dir);
    while ((ent = readdir(dir)) != NULL)
      printf("%s\n",ent->d_name);
    if (closedir(dir) != 0)
      perror("Unable to close directory");
}

void main(int argc,char *argv[])
{
    if (argc != 2)
    {
      printf("usage: opendir dirname\n");
      exit(1);
    }
    scandir(argv[1]);
    exit(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
opendir	+	+		
wopendir				

1.1.2.8.3 readdir, wreaddir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
struct dirent *readdir(DIR *dirp);
```

```
struct wdirent *wreaddir(wDIR *dirp)
```

Description

Reads the current entry from a directory stream.

readdir is available on POSIX-compliant UNIX systems.

The readdir function reads the current directory entry in the directory stream pointed to by dirp. The directory stream is advanced to the next entry.

The readdir function returns a pointer to a **dirent** structure that is overwritten by each call to the function on the same directory stream. The structure is not overwritten by a readdir call on a different directory stream.

The **dirent** structure corresponds to a single directory entry. It is defined in dirent.h and contains (in addition to other non-accessible members) the following member:

char d_name[];

where d_name is an array of characters containing the null-terminated file name for the current directory entry. The size of the array is indeterminate; use strlen to determine the length of the file name.

All valid directory entries are returned, including subdirectories, “.” and “..” entries, system files, hidden files, and volume labels. Unused or deleted directory entries are skipped.

A directory entry can be created or deleted while a directory stream is being read, but readdir might or might not return the affected directory entry. Rewinding the directory with rewinddir or reopening it with opendir ensures that readdir will reflect the current state of the directory.

The wreaddir function is the Unicode version of readdir. It uses the **wdirent** structure but otherwise is similar to readdir.

Return Value

On success, readdir returns a pointer to the current directory entry for the directory stream.

If the end of the directory has been reached, or dirp does not refer to an open directory stream, readdir returns NULL.

Portability

	POSIX	Win32	ANSI C	ANSI C++
readdir	+	+		
wreaddir		+		

1.1.2.8.4 rewinddir, wrewinddir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
void rewinddir(DIR *dirp);
```

```
void wrewinddir(wDIR *dirp);
```

Description

Resets a directory stream to the first entry.

rewinddir is available on POSIX-compliant UNIX systems.

The rewinddir function repositions the directory stream dirp at the first entry in the directory. It also ensures that the directory stream accurately reflects any directory entries that might have been created or deleted since the last opendir or rewinddir on that directory stream.

wrewinddir is the Unicode version of rewinddir.

Return Value

None.

Example

```
/* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
```

```
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
 }

 void main(int argc,char *argv[])
 {
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
rewind	+	+		
wrewinddir		+		

1.1.2.9 dos.h

The following functions, macros, and classes are provided in `dos.h`:

1.1.2.9.1 disable, _disable, enable, _enable

Header File

dos.h

Category

Miscellaneous Routines

Prototype

```
void disable(void);
```

```
void _disable(void);
```

```
void enable(void);
void _enable(void);
```

Description

Disables and enables interrupts.

These macros are designed to provide a programmer with flexible hardware interrupt control.

disable and _disable macros disable interrupts. Only the NMI (non-maskable interrupt) is allowed from any external device.

enable and _enable macros enable interrupts, allowing any device interrupts to occur.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.9.2 dostounix

Header File

dos.h

Category

Time and Date Routines

Prototype

```
long dostounix(struct date *d, struct time *t);
```

Description

Converts date and time to UNIX time format.

dostounix converts a date and time as returned from getdate and gettime into UNIX time format. d points to a date structure, and t points to a time structure containing valid date and time information.

The date and time must not be earlier than or equal to Jan 1 1980 00:00:00.

Return Value

Returns UNIX version of current date and time parameters: number of seconds since 00:00:00 on January 1, 1970 (GMT).

Example

```
#include <time.h>
#include <stddef.h>
#include <dos.h>
#include <stdio.h>
int main(void)
{
  time_t t;
  struct time d_time;
  struct date d_date;
  struct tm *local;
  getdate(&d_date);
  gettime(&d_time);
```

```
  t = dostounix(&d_date, &d_time);
  local = localtime(&t);
  printf("Time and Date: %s\n", asctime(local));
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.9.3 geninterrupt

Header File

Category

Prototype

```
void geninterrupt(int intr_num);
```

Description

Return Value

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.9.4 getdate, setdate

Header File

dos.h

Category

Time and Date Routines

Prototype

```
void getdate(struct date *datep);
```

```
void setdate(struct date *datep);
```

Description

Gets and sets system date.

getdate fills in the date structure (pointed to by datep) with the system's current date.

setdate sets the system date (month, day, and year) to that in the date structure pointed to by datep. Note that a request to set a date might fail if you do not have the privileges required by the operating system.

The date structure is defined as follows:

```
struct date{
```

```
int da_year; /* current year */
```

```
char da_day; /* day of the month */
```

```
char da_mon; /* month (1 = Jan) */

};
```

Return Value

getdate and setdate do not return a value.

Example

```
#include <dos.h>
#include <stdio.h>
int main(void)
{
   struct date d;
   getdate(&d);
   printf("The current year is: %d\n", d.da_year);
   printf("The current day is: %d\n", d.da_day);
   printf("The current month is: %d\n", d.da_mon);
   return 0;
}
```

1.1.2.9.5 getdfree

Header File

dos.h

Category

Directory Control Routines, Miscellaneous Routines

Prototype

```
void getdfree(unsigned char drive, struct dfree *dtable);
```

Description

Gets disk free space.

getdfree accepts a drive specifier in drive (0 for default, 1 for A, and so on) and fills the dfree structure pointed to by dtable with disk attributes.

The dfree structure is defined as follows:

```
struct dfree {

unsigned df_avail; /* available clusters */

unsigned df_total; /* total clusters */

unsigned df_bsec; /* bytes per sector */

unsigned df_sclus; /* sectors per cluster */

};
```

Return Value

getdfree returns no value. In the event of an error, df_sclus in the dfree structure is set to (**unsigned**) -1.

Example

```
#include <stdio.h>
#include <dos.h>
#include <process.h>
int main(void)
{
```

```
  struct dfree free;
  long avail;
  getdfree(0, &free);
  if ( free.df_sclus == -1)
  {
    printf("Error in getdfree() call\n");
    exit(1);
  }
  avail = (long) free.df_avail
        * (long) free.df_bsec
        * (long) free.df_sclus;
  printf("The current drive has %ld bytes available\n", avail);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.9.6 _getdrive

Header File

dos.h

Category

Directory Control Routines

Prototype

```
int _getdrive(void);
```

Description

Gets the current drive.

_getdrive gets the the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on.

Return Value

_getdrive returns the current drive number on success or -1 in the event of error.

Example

```
#include <stdio.h>
#include <direct.h>
int main(void)
{
  int disk;
  disk = _getdrive() + 'A' - 1;
  printf("The current drive is: %c\n", disk);
  return 0;
}
```

1.1.2.9.7 gettime, settime

Header File

dos.h

Category

Time and Date Routines

Prototype

```
void gettime(struct time *timep);
void settime(struct time *timep);
```

Description

Gets and sets the system time.

gettime fills in the time structure pointed to by timep with the system's current time.

- settime sets the system time to the values in the time structure pointed to by timep.

The time structure is defined as follows:

```
struct time {
unsigned char ti_min; /* minutes */
unsigned char ti_hour; /* hours */
unsigned char ti_hund; /* hundredths of seconds */
unsigned char ti_sec; /* seconds */
};
```

Return Value

None.

Example

```
#include   <stdio.h>
#include   <dos.h>
int main(void)
{
   struct  time t;
   gettime(&t);
   printf("The current time is: %2d:%02d:%02d.%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
   return 0;
}
```

1.1.2.9.8 _sleep

Header File

dos.h

Category

Process Control Routines

Prototype

```
void _sleep(unsigned seconds);
```

Description

Suspends execution for an interval (seconds).

With a call to _sleep, the current program is suspended from execution for the number of seconds specified by the argument seconds. The interval is accurate only to the nearest hundredth of a second or to the accuracy of the operating system clock, whichever is less accurate.

Return Value

None.

Example

```
#include <dos.h>
#include <stdio.h>
int main(void)
{
   int i;
   for (i=1; i<5; i++)
   {
      printf("Sleeping for %d seconds\n", i);
      _sleep(i);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.9.9 unixtodos

Header File

dos.h

Category

Time and Date Routines

Prototype

```
void unixtodos(long time, struct date *d, struct time *t);
```

Description

Converts date and time from UNIX to DOS format.

unixtodos converts the UNIX-format time given in time to DOS format and fills in the date and time structures pointed to by d and t.

time must not represent a calendar time earlier than Jan. 1, 1980 00:00:00.

Return Value

None.

Example

```
#include <stdio.h>
#include <dos.h>
char *month[] = {"---", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
#define SECONDS_PER_DAY 86400L  /* the number of seconds in one day */
struct date dt;
struct time tm;
int main(void)
{
   unsigned long val;
/* get today's date and time */
```

```
   getdate(&dt);
   gettime(&tm);
   printf("today is %d %s %d\n", dt.da_day, month[dt.da_mon], dt.da_year);
/*convert date and time to unix format (num of seconds since Jan 1, 1970*/
   val = dostounix(&dt, &tm);
/* subtract 42 days worth of seconds */
   val -= (SECONDS_PER_DAY * 42);
/* convert back to dos time and date */
   unixtodos(val, &dt, &tm);
   printf("42 days ago it was %d %s %d\n",
          dt.da_day, month[dt.da_mon], dt.da_year);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.9.10 _unlink, _wunlink

Header File

dos.h

Category

Input/output Routines

Prototype

```
int _unlink(const char *filename);
```

```
int _wunlink(const wchar_t *filename);
```

Description

Deletes a file.

_unlink deletes a file specified by filename. Any drive, path, and file name can be used as a filename. Wildcards are not allowed.

Read-only files cannot be deleted by this call. To remove read-only files, first use chmod or _rtl_chmod to change the read-only attribute.

Note: If the file is open, it must be closed before unlinking it.

_wunlink is the Unicode version of _wunlink. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform identically.

Return Value

On success, _unlink returns 0.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Example

```
#include <stdio.h>
#include <io.h>
```

```
int main(void)
{
   FILE *fp = fopen("junk.jnk","w");
   int status;
   fprintf(fp,"junk");
   status = access("junk.jnk",0);
   if (status == 0)
      printf("File exists\n");
   else
      printf("File doesn't exist\n");
   fclose(fp);
   unlink("junk.jnk");
   status = access("junk.jnk",0);
   if (status == 0)
      printf("File exists\n");
   else
      printf("File doesn't exist\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_unlink	+	+		
_wunlink		NT only		

1.1.2.9.11 _osmajor

Header File

dos.h

Syntax

```
extern unsigned char _osmajor;
```

Description

The major version number of the operating system is available individually through _osmajor. For example, if you are running DOS version 3.2, _osmajor will be 3.

This variable can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew and _rtl_open.

1.1.2.9.12 _osminor

Header File

dos.h

Syntax

```
extern unsigned char _osminor;
```

Description

The minor version number of the operating system is available individually through _osminor. For example, if you are running DOS version 3.2, _osminor will be 20.

This variables can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines

behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew and _rtl_open.

1.1.2.9.13 _osversion

Header File

dos.h

Syntax

```
extern unsigned _osversion;
```

Description

_osversion contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the x is the major version number, and y is the minor version number.)

_osversion is functionally identical to _version.

1.1.2.9.14 _version

Header File

dos.h

Syntax

```
extern unsigned _version;
```

Description

_version contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the x is the major version number, and y is the minor version number.)

1.1.2.9.15 FA_xxxx #defines

Header File

dos.h

Description

File attributes

Constant	Description
FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

1.1.2.9.16 NFDS #define

Header File

dos.h

Description

Maximum number of file descriptors.

1.1.2.10 errno.h

The following functions, macros, and classes are provided in `errno.h`:

1.1.2.10.1 perror,_wperror

Header File

errno.h, stdio.h

Category

Diagnostic Routines, Input/output Routines

Prototype

```
void perror(const char *s);
void _wperror(const wchar_t *s);
```

Description

Prints a system error message.

perror prints to the stderr stream (normally the console) the system error message for the last library routine that set the global variable errno.

It prints the argument s followed by a colon (:) and the message corresponding to the current value of the global variable errno and finally a new line. The convention is to pass the file name of the program as the argument string.

The array of error message strings is accessed through the global variable _sys_errlist. The global variable errno can be used as an index into the array to find the string corresponding to the error number. None of the strings include a newline character.

The global variable _sys_nerr contains the number of entries in the array.

The following messages are generated by perror:

Note: For Win32 GUI applications, stderr must be redirected.

Arg list too big

Attempted to remove current directory

Bad address

Bad file number

Block device required

Broken pipe

Cross-device link

Error 0

Exec format error

Executable file in use

File already exists

File too large

Illegal seek

Inappropriate I/O control operation

Input/output error

Interrupted function call

Invalid access code

Invalid argument Resource busy

Invalid dataResource temporarily unavailable

Invalid environment

Invalid format

Invalid function number

Invalid memory block address

Is a directory

Math argument

Memory arena trashed

Name too long

No child processes

No more files

No space left on device

No such device

No such device or address

No such file or directory

No such process

Not a directory

Not enough memory

Not same device

Operation not permitted

Path not found

Permission denied

Possible deadlock

Read-only file system

Resource busy

Resource temporarily unavailable

Result too large

Too many links

Too many open files

Example

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    fp = fopen("perror.dat", "r");
    if (!fp)
       perror("Unable to open file for reading");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
perror	+	+	+	+
_wperror		+		

1.1.2.10.2 errno (C Runtime Library Reference)

Header File

errno.h

Syntax

```
extern int errno;
```

Description

errno is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

When an error in a math or system call occurs, errno is set to indicate the type of error. Sometimes errno and _doserrno are equivalent. At other times, errno does not contain the actual operating system error code, which is contained in _doserrno. Still other errors might occur that set only errno, not _doserrno.

1.1.2.10.3 _doserrno

Header File

errno.h

Syntax

```
extern int _doserrno;
```

Description

_doserrno is a variable that maps many operating system error codes to errno; however, perror does not use _doserrno directly.

When an operating system call results in an error, _doserrno is set to the actual operating system error code. errno is a parallel error variable inherited from UNIX.

The following list gives mnemonics for the actual DOS error codes to which _doserrno can be set. (This value of _doserrno may or may not be mapped (through errno) to an equivalent error message string in _sys_errlist.

E2BIG	Bad environ
EACCES	Access denied
EACCES	Bad access
EACCES	Is current dir
EBADF	Bad handle
EFAULT	Reserved
EINVAL	Bad data
EINVAL	Bad function
EMFILE	Too many open
ENOENT	No such file or directory
ENOEXEC	Bad format
ENOMEM	Mcb destroyed
ENOMEM	Out of memory
ENOMEM	Bad block
EXDEV	Bad drive
EXDEV	Not same device

1.1.2.10.4 _sys_errlist

Header File

errno.h

Syntax

```
extern char * _sys_errlist[ ];
```

Description

_sys_errlist is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

To provide more control over message formatting, the array of message strings is provided in _sys_errlist. You can use errno as an index into the array to find the string corresponding to the error number. The string does not include any newline character.

Example

```
printf("%s\n", _sys_errlist[ENOPATH]);
```

This code statement that uses the Mnemonic ENOPATH will output the string "Path not found".

The following table gives mnemonics and their meanings for the values stored in _sys_errlist. The list is alphabetically ordered for ease your reading convenience. For the numerical ordering, see the header file errno.h.

1.1.2.10.5 _sys_nerr

Header File

errno.h

Syntax

```
extern int _sys_nerr;
```

Description

_sys_nerr is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

This variable is defined as the number of error message strings in _sys_errlist.

1.1.2.10.6 EDOM, ERANGE, #defines

Header File

errno.h, math.h

Description

These are the mnemonics and meanings for the error numbers found in math.h and errno.

Name	Meaning
EDOM	Error code for math domain error
ERANGE	Error code for result out of range

1.1.2.10.7 Error Numbers in errno

Header File

errno.h

Description

These are the mnemonics and meanings for the error numbers found in errno.

Each value listed can be used to index into the sys_errlist array for displaying messages.

Also, perror will display messages.

Mnemonic	Meaning
EZERO	Error 0
EINVFNC	Invalid function number
ENOFILE	File not found
ENOPATH	Path not found
ECONTR	Memory blocks destroyed
EINVMEM	Invalid memory block address
EINVENV	Invalid environment
EINVFMT	Invalid format

EINVACC	Invalid access code
EINVDAT	Invalid data
EINVDRV	Invalid drive specified
ECURDIR	Attempt to remove CurDir
ENOTSAM	Not same device
ENMFILE	No more files
ENOENT	No such file or directory
EMFILE	Too many open files
EACCES	Permission denied
EBADF	Bad file number
ENOMEM	Not enough memory
EFAULT	Unknown error
ENODEV	No such device
EINVAL	Invalid argument
E2BIG	Arg list too long
ENOEXEC	Exec format error
EXDEV	Cross-device link
ENFILE	Too many open files
ECHILD	No child process
ENOTTY	Terminal control function attempted on a file that is not a terminal. (POSIX – Not used in Win32 applications.)
ETXTBSY	Not used in Win32 applications
EFBIG	An attempt was made to write to a file, beyond the maximum file size. (POSIX – Not used in Win32 applications.)
ESOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	The number of links exceeds LINK_MAX. (POSIX – Not used in Win32 applications.)
EPIPE	Broken pipe
EDOM	Math argument
ERANGE	Result too large
EEXIST	File already exists
EDEADLOCK	Locking violation
EPERM	Operation not permitted
ESRCH	No such process id. (POSIX – Not used in Win32 applications.)
EINTR	Interrupted function call
EIO	Input/output error
ENXIO	No such device or address
EAGAIN	Resource temporarily unavailable

ENOTBLK	Not used in Win32 applications
EBUSY	Resource busy
ENOTDIR	A pathname component is not a directory. (POSIX – Not used in Win32 applications.)
EISDIR	An attempt was made to open a directory for writing, or to rename a file with the same name as an existing directory. (POSIX – Not used in Win32 applications.)
EUCLEAN	Not used in Win32 console applications

1.1.2.11 except.h

The following functions, macros, and classes are provided in `except.h`:

1.1.2.11.1 set_terminate

Header File

except.h

Syntax

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler t_func);
```

Description

set_terminate lets you install a function that defines the program's termination behavior when a handler for the exception cannot be found. The actions are defined in t_func, which is declared to be a function of type terminate_handler. A terminate_handler type, defined in except.h, is a function that takes no arguments, and returns void.

By default, an exception for which no handler can be found results in the program calling the terminate function. This will normally result in a call to abort. The program then ends with the message Abnormal program termination. If you want some function other than abort to be called by the terminate function, you should define your own t_func function. Your t_func function is installed by set_terminate as the termination function. The installation of t_func lets you implement any actions that are not taken by abort.

Return Value

The previous function given to set_terminate will be the return value.

The definition of t_func must terminate the program. Such a user-defined function must not return to its caller, the terminate function. An attempt to return to the caller results in undefined program behavior. It is also an error for t_func to throw an exception.

1.1.2.11.2 set_unexpected

Header File

except.h

Syntax

```
typedef void ( * unexpected_handler )();
unexpected_handler set_unexpected(unexpected_handler unexpected_func);
```

Description

set_unexpected lets you install a function that defines the program's behavior when a function throws an exception not listed in its exception specification. The actions are defined in unexpected_func, which is declared to be a function of type unexpected_handler. An unexpected_handler type, defined in except.h, is a function that takes no arguments, and returns void.

By default, an unexpected exception causes unexpected to be called. If is defined, it is subsequently called by unexpected. Program control is then turned over to the user-defined unexpected_func. Otherwise, terminate is called.

Return Value

The previous function given to set_unexpected will be the return value.

The definition of unexpected_func must not return to its caller, the unexpected function. An attempt to return to the caller results in undefined program behavior.

unexpected_func can also call abort, exit, or terminate.

1.1.2.11.3 terminate

Header File

except.h

Syntax

```
void terminate();
```

Description

The function terminate can be called by unexpected or by the program when a handler for an exception cannot be found. The default action by terminate is to call abort. Such a default action causes immediate program termination.

You can modify the way that your program will terminate when an exception is generated that is not listed in the exception specification. If you do not want the program to terminate with a call to abort, you can instead define a function to be called. Such a function (called a terminate_handler) will be called by terminate if it is registered with set_terminate.

Return Value

None.

1.1.2.11.4 unexpected

Header File

except.h

Syntax

```
void unexpected();
```

Description

The unexpected function is called when a function throws an exception not listed in its exception specification. The program calls unexpected, which by default calls any user-defined function registered by set_unexpected. If no function is registered with set_unexpected, the unexpected function then calls terminate.

Return Value

None, although unexpected may throw an exception.

1.1.2.11.5 __throwExceptionName

Header File

except.h

Syntax

```
extern char * _RTLENTRY __ThrowExceptionName();
#define __throwExceptionName __ThrowExceptionName()
```

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

1.1.2.11.6 __throwFileName

Header File

except.h

Syntax

```
extern char * _RTLENTRY __ThrowFileName();
#define __throwFileName __ThrowFileName()
```

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

To get the file name for a thrown exception with __throwFileName, you must compile the module with the -xp compiler option.

1.1.2.11.7 __throwLineNumber

Header File

except.h

Syntax

```
extern unsigned _RTLENTRY __ThrowLineNumber();
#define __throwLineNumber __ThrowLineNumber()
```

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

To get the line number for a thrown exception with __throwLineNumber, you must compile the module with the -xp compiler option.

1.1.2.12 fastmath.h

The following functions, macros, and classes are provided in `fastmath.h`:

1.1.2.12.1 Using fastmath math routines

Header File

fastmath.h

Category

Math Routines

Description

The FastMath routines are high performance math routines that don't check for most error conditions and never call matherr. They are coded for maximum speed. These functions are never exported from the RTLDLL, which means that they always get linked directly into the PE file that is being created.

When you include fastmath.h, the following math functions are remapped to these fastmath functions.

Math Routine	Fastmath Routine	Math Routine	Fastmath Routine
acos	_fm_acos	asin	_fm_asin
atan	_fm_atan	atan2	_fm_atan2
cos	_fm_cos	cosh	_fm_cosh
exp	_fm_exp	fabs	_fm_fabs
asin	_fm_asin	atan	_fm_atan
atan2	_fm_atan2	cos	_fm_cos
cosh	_fm_cosh	exp	_fm_exp
fabs	_fm_fabs	frexp	_fm_frexp
hypot	_fm_hypot	ldexp	_fm_ldexp
log	_fm_log	log10	_fm_log10
sin	_fm_sin	sinh	_fm_sinh
sqrt	_fm_sqr	tan	_fm_tan
tanh	_fm_tanh	sincos	_fm_sincos
acosl	_fm_acosl	asinl	_fm_asinl
atan2l	_fm_atan2l	atanl	_fm_atanl
coshl	_fm_coshl	cosl	_fm_cosl
expl	_fm_expl	fabsl	_fm_fabsl
frexpl	_fm_frexpl	hypotl	_fm_hypotl
ldexpl	_fm_ldexpl	log10l	_fm_log10l
logl	_fm_logl	sinhl	_fm_sinhl
sinl	_fm_sinl	sqrtl	_fm_sqrtl
tanhl	_fm_tanhl	tanl	_fm_tanl
sincosl	_fm_sincosl	atanhl	_fm_atanhl
acoshl	_fm_acoshl	asinhl	_fm_asinhl

If you don't want the standard C function names remapped to the FastMath versions, then define _FM_NO_REMAP. The FastMath routines can still be called with their _fm_xxx names.

The following additional functions are available in FastMath; they are not directly supported in the regular RTL:

```
void _FMAPI _fm_sincos(double __a, double *__x, double *__y);
void _FMAPI _fm_sincosl(long double __a, long double *__x, long double *__y);
long double _FMAPI _fm_atanhl (long double __x);
long double _FMAPI _fm_acoshl (long double __x);
long double _FMAPI _fm_asinhl (long double __x);
__inline void _fm_fwait(void)
unsigned int _FMAPI _fm_init(void);
```

_fm_fwait is a special inline function that performs an intrinsic FWAIT instruction.

_Fm_init is a function that can be called to mask all fpu exceptions prior to using the FastMath routines.

1.1.2.13 fcntl.h

The following functions, macros, and classes are provided in `fcntl.h`:

1.1.2.13.1 open, _wopen

Header File

io.h, fcntl.h

Category

Input/output Routines

Prototype

```
int open(const char *path, int access [, unsigned mode]);
int _wopen(const wchar_t *path, int access [, unsigned mode]);
```

Description

Opens a file for reading or writing.

open opens the file specified by path, then prepares it for reading and/or writing as determined by the value of access.

To create a file in a particular mode, you can either assign to the global variable _fmode or call open with the O_CREAT and O_TRUNC options ORed with the translation mode desired.

```
open("XMP",O_CREAT|O_TRUNC|O_BINARY,S_IREAD)
```

creates a binary-mode, read-only file named XMP, truncating its length to 0 bytes if it already existed.

For open, access is constructed by bitwise ORing flags from the following lists. Only one flag from the first list can be used (and one must be used); the remaining flags can be used in any logical combination.

These symbolic constants are defined in fcntl.h.

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_CREAT	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of mode are used to set the file attribute bits as in chmod.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	Can be given to explicitly open the file in binary mode.
O_TEXT	Can be given to explicitly open the file in text mode.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable _fmode.

If the O_CREAT flag is used in constructing access, you need to supply the mode argument to open from the following symbolic constants defined in sys\stat.h.

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read and write

Return Value

On success, open returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, open returns -1 and the global variable errno is set to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	No such file or directory

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = open("TEST.$$$", O_CREAT | O_TEXT)) == -1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   close(handle);
```

```
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
open	+	+		
_wopen		NT only		

1.1.2.13.2 _pipe

Header File

io.h, fcntl.h

Category

Input/output Routines

Syntax

```
int _pipe(int *handles, unsigned int size, int mode);
```

Description

Creates a read/write pipe.

The _pipe function creates an anonymous pipe that can be used to pass information between processes. The pipe is opened for both reading and writing. Like a disk file, a pipe can be read from and written to, but it does not have a name or permanent storage associated with it; data written to and from the pipe exist only in a memory buffer managed by the operating system.

The read handle is returned to handles[0], and the write handle is returned to handles[1]. The program can use these handles in subsequent calls to read, write, dup, dup2, or close. When all pipe handles are closed, the pipe is destroyed.

The size of the internal pipe buffer is size. A recommended minimum value is 512 bytes.

The translation mode is specified by mode, as follows:

O_BINARY	The pipe is opened in binary mode
O_TEXT	The pipe is opened in text mode

If mode is zero, the translation mode is determined by the external variable _fmode.

Return Value

On success, _pipe returns 0 and returns the pipe handles to handles[0] and handles[1].

On error, it returns -1 and sets errno to one of the following values:

EMFILE	Too many open files
ENOMEM	Out of memory

Example

```
/*
 There are two short programs here.  SEND spawns a child
 process, RECEIVE.  Each process holds one end of a
 pipe.  The parent transmits its command-line argument
 to the child, which prints the string and exits.
 IMPORTANT:  The parent process must be linked with
```

```
 the \32bit\fileinfo.obj file.  The code in fileinfo
 enables a parent to share handles with a child.
 Without this extra information, the child cannot use
 the handle it receives.
*/
/* SEND */
#include <fcntl.h>           // _pipe()
#include <io.h>            // write()
#include <process.h>          // spawnl() cwait()
#include <stdio.h>          // puts() perror()
#include <stdlib.h>          // itoa()
#include <string.h>          // strlen()
#define DECIMAL_RADIX 10      // for atoi()
enum PIPE_HANDLES { IN, OUT };  // to index the array of handles
int main(int argc, char *argv[])
{
  int handles[2];           // in- and
//outbound pipe handles
  char handleStr[10];        // a handle
//stored as a string
  int pid;
  // system's ID for child process
  if (argc <= 1)
  {
    puts("No message to send.");
    return(1);
  }
  if (_pipe(handles, 256, O_TEXT) != 0)
  {
    perror("Cannot create the pipe");
    return(1);
  }
  // store handle as a string for passing on the command line
  itoa(handles[IN], handleStr, DECIMAL_RADIX);
  // create the child process, passing it the inbound pipe handle
     spawnl(P_NOWAIT, "receive.exe", "receive.exe", handleStr, NULL);
  // transmit the message
  write(handles[OUT], argv[1], strlen(argv[1])+1);
  // when done with the pipe, close both handles
  close(handles[IN]);
  close(handles[OUT]);
  // wait for the child to finish
  wait(NULL);
  return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.13.3 _sopen, _wsopen

Header File

fcntl.h, sys\stat.h, share.h, io.h, stdio.h

Category

Input/output Routines

Prototype

```
int _sopen(char *path, int access, int shflag[, int mode]);
```

```
int _wsopen(wchar_t *path, int access, int shflag[, int mode]);
```

Description

Opens a shared file.

_sopen opens the file given by path and prepares it for shared reading or writing, as determined by access, shflag, and mode.

_wsopen is the Unicode version of _sopen. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform identically.

For _sopen, access is constructed by ORing flags bitwise from the following lists:

Read/write flags

You can use only one of the following flags:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Other access flags

You can use any logical combination of the following flags:

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer is set to the end of the file prior to each write.
O_CREA	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of mode are used to set the file attribute bits as in chmod.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	This flag can be given to explicitly open the file in binary mode.
O_TEXT	This flag can be given to explicitly open the file in text mode.
O_NOINHERIT	The file is not passed to child programs.

Note: These O_... symbolic constants are defined in fcntl.h.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable _fmode.

If the O_CREAT flag is used in constructing access, you need to supply the mode argument to _sopen from the following symbolic constants defined in sys\stat.h.

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read/write

shflag specifies the type of file-sharing allowed on the file path. Symbolic constants for shflag are defined in share.h.

SH_COMPAT	Sets compatibility mode.
SH_DENYRW	Denies read/write access
SH_DENYWR	Denies write access
SH_DENYRD	Denies read access

SH_DENYNONE	Permits read/write access
SH_DENYNO	Permits read/write access

Return Value

On success, _sopen returns a nonnegative integer (the file handle), and the file pointer (that marks the current position in the file) is set to the beginning of the file.

On error, it returns -1, and the global variable errno is set to

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	Path or file function not found

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   int handle,
       handle1;
   handle = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYWR, S_IREAD);
   if      (handle == -1)
   {
     perror (sys_errlist[errno]);
     exit (1);
   }
   if (!handle)
   {
      printf("sopen failed\n");
      exit(1);
   }
   /*      Attempt sopen for write.
   */
   handle1 = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYWR, S_IREAD);
   if      (handle1 == -1)
   {
     perror (sys_errlist[errno]);
     exit (1);
   }
   if (!handle1)
   {
      printf("sopen failed\n");
      exit(1);
   }
   close (handle);
   close (handle1);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_sopen		+		
_wsopen		+		

1.1.2.13.4 _fmode

Header File

fcntl.h

Syntax

```
extern int _fmode;
```

Description

_fmode determines in which mode (text or binary) files will be opened and translated. The value of _fmode is O_TEXT by default, which specifies that files will be read in text mode. If _fmode is set to O_BINARY, the files are opened and read in binary mode. (O_TEXT and O_BINARY are defined in fcntl.h.)

In text mode, carriage-return/linefeed (CR/LF) combinations are translated to a single linefeed character (LF) on input. On output, the reverse is true: LF characters are translated to CR/LF combinations.

In binary mode, no such translation occurs.

You can override the default mode as set by _fmode by specifying a t (for text mode) or b (for binary mode) in the argument type in the library functions fopen, fdopen, and freopen. Also, in the function open, the argument access can include either O_BINARY or O_TEXT, which will explicitly define the file being opened (given by the path argument to the open function) to be in either binary or text mode.

1.1.2.13.5 O_xxxx #defines

Header File

fcntl.h

Description

These #defines are bit definitions for a file-access argument.

These RTL file-open functions use some (not all) of these definitions:

- fdopen
- fopen
- freopen
- _fsopen
- open
- _rtl_open
- sopen

sopen also uses file-sharing symbolic constants in the file-access argument.

	Constant	Description
Read/Write flag		
	O_RDONLY	Open for reading only
	O_WRONLY	Open for writing only
	O_RDWR	Open for reading and writing
Other access flags		
	O_NDELAY	Not used; for UNIX compatibility.
	O_APPEND	Append to end of file
		If set, the file pointer is set to the end of the file prior to each write.
	O_CREAT	Create and open file
		If the file already exists, has no effect.
		If the file does not exist, the file is created.
	O_EXCL	Exclusive open: Used only with O_CREAT.
		If the file already exists, an error is returned.
	O_TRUNC	Open with truncation
		If the file already exists, its length is truncated to 0. The file attributes remain unchanged.
Binary-mode/Text-mode flags		
	O_BINARY	No translation: Explicitly opens the file in binary mode
	O_TEXT	CR-LF translation: Explicitly opens the file in text mode
Additional values available using _rtl_open		
	O_NOINHERIT	Child processes inherit file
	O_DENYALL	Error if opened for read/write
	O_DENYWRITE	Error if opened for write
	O_DENYREAD	Error if opened for read
	O_DENYNONE	Allow concurrent access

Note: Only one of the O_DENYxxx options can be included in a single open. These file-sharing attributes are in addition to any locking performed on the files.

Do not modify		
	O_CHANGED	Special DOS read-only bit
	O_DEVICE	Special DOS read-only bit

1.1.2.14 float.h

The following functions, macros, and classes are provided in `float.h`:

1.1.2.14.1 _chgsign, _chgsignl

Header File

float.h

Category

Math Routintes

Prototype

```
double _chgsign(double d);
long double _chgsignl(long double ld);
```

Description

Reverses the sign of a double-precision floating-point argument, d.

_chgsignl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns a value of the same magnitude and exponent as the argument, but with the opposite sign. There is no error return value.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_chgsign		+		
_chgsignl		+		

1.1.2.14.2 _clear87, _clearfp

Header File

float.h

Category

Math Routines

Prototype

```
unsigned int _clear87 (void);
unsigned int _clearfp (void);
```

Description

Clears the floating-point status word.

_clear87 clears the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

_clearfp is identical to _clear87 and is for Microsoft compatibility.

Return Value

The bits in the value returned indicate the floating-point status before it was cleared. For information on the status word, refer to the constants defined in float.h.

Example

```
#include <stdio.h>
#include <float.h>
int main(void)
{
   float x;
   double y = 1.5e-100;
   printf("\nStatus 87 before error: %X\n", _status87());
   x = y; /* create underflow and precision loss */
   printf("Status 87 after  error: %X\n", _status87());
   _clear87();
   printf("Status 87 after  clear: %X\n", _status87());
   y = x;
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.14.3 _control87, _controlfp

Header File

float.h

Category

Math Routines

Prototype

```
unsigned int _control87(unsigned int newcw, unsigned int mask);
```

```
unsigned int _controlfp(unsigned int newcw, unsigned int mask);
```

Description

Manipulates the floating-point control word.

_control87 retrieves or changes the floating-point control word.

The floating-point control word is an **unsigned int** that, bit by bit, specifies certain modes in the floating-point package; namely, the precision, infinity, and rounding modes. Changing these modes lets you mask or unmask floating-point exceptions.

_control87 matches the bits in mask to the bits in newcw. If a mask bit equals 1, the corresponding bit in newcw contains the new value for the same bit in the floating-point control word, and _control87 sets that bit in the control word to the new value.

Here is a simple illustration:

Original control word:	0100	0011	0110	0011	
mask:		1000	0001	0100	1111
newcw:		1110	1001	0000	0101

Changing bits:		1xxx	xxx1	x0xx	0101

If mask equals 0, _control87 returns the floating-point control word without altering it.

_controlfp is for Microsoft compatibility. _controlfp is identical to _control87 except that it always removes (turns off) the EM_DEMORMAL bit from the mask parameter.

Return Value

The bits in the value returned reflect the new floating-point control word. For a complete definition of the bits returned by _control87, see the header file float.h.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_control87		+		
_controlfp		+		

1.1.2.14.4 _copysign, _copysignl

Header File

float.h

Category

Math Routines

Prototype

```
double _copysign(double da, double db);
long double _copysignl(long double lda, long double ldb);
```

Description

Returns the double-precision floating point argument da, with the same sign as the double-precision floating-point argument db.

_copysignl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns the first value with the same magnitude and exponent, but with the sign of the second value. There is no error value returned.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_copysign		+		
_copysignl		+		

1.1.2.14.5 _finite, _finitel

Header File

float.h

Category

Math Routines

Prototype

```
int _finite(double d);
int _finitel(long double ld);
```

Description

Determines whether a given double-precision floating point value d is finite.

_finitel is the **long double** version; it takes a **long double** argument.

Return Value

Returns non-zero if the argument is finite, and 0 if it is not.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_finite		+		
_finitel		+		

1.1.2.14.6 _fpclass, _fpclassl

Header File

float.h

Category

Math Routines

Prototype

```
int _fpclass(double d);
int _fpclassl(long double ld);
```

Description

Returns an integer value representing the type (class) of an IEEE real for doubles. This value contains information on the floating-point class of the argument.

_fpclassl is the **long double** version; it takes a **long double** argument and returns the type (class) of an IEEE real for long doubles.

Return Value

Returns an integer value that indicates the floating-point class of its argument. The possible values, which are listed in the table below, are defined in FLOAT.H.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fpclass		+		
_fpclassl		+		

1.1.2.14.7 _fpreset

Header File

float.h

Category

Math Routines

Prototype

```
void _fpreset(void);
```

Description

Reinitializes floating-point math package.

_fpreset reinitializes the floating-point math package. This function is usually used in conjunction with system or the exec... or spawn... functions. It is also used to recover from floating-point errors before calling longjmp.

Note: If an 80x87 coprocessor is used in a program a child process (executed by the system, or by an exec... or spawn... function) might alter the parent process' floating-point state.

- If you use an 80x87 take the following precautions:
- Do not call system or an exec... or spawn... function while a floating-point expression is being evaluated.

Call _fpreset to reset the floating-point state after using system exec... or spawn... if there is any chance that the child process performed a floating-point operation with the 80x87.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.14.8 _isnan, _isnanl

Header File

float.h

Category

Classification Routines, Math Routines

Prototype

```
int _isnan(double d);
int _isnanl(long double ld);
```

Description

Tests whether a given double-precision floating-point value d is a NaN.

_isnanl is the **long double** version; it takes a **long double** argument.

Return Value

Returns a nonzero value (TRUE) if the value passed in is a NaN; otherwise it returns 0 (FALSE). The non-zero return value corresponds to either _FPCLASS_SNAN, if the NaN is of the signaling type, or _FPCLASS_QNAN, if the NaN is of the quiet type. The values for _FPCLASS_SNAN and _FPCLASS_QNAN are in float.h.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_isnan		+		
_isnanl		+		

1.1.2.14.9 _logb, _logbl

Header File

float.h

Category

Math Routines

Prototype

```
double _logb(double d);
long double _logbl(long double ld);
```

Description

Extracts the exponential value of a double-precision floating-point argument. If the argument is denormalized, it is treated as if it were normalized.

_logbl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns the unbiased exponent of the value passed in.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_logb		+		
_logbl		+		

1.1.2.14.10 _nextafter, _nextafterl

Header File

float.h

Category

Math Routines

Prototype

```
double _nextafter(double da, double db);
```

```
long double _nextafterl(long double lda, long double ldb);
```

Description

Takes two doubles (da and db) and returns the closest representable neighbor of da in the direction toward db.

If da == db, _nextafter returns da, with no exception triggered. If either da or db is a quiet NaN, then the return value is one or the other of the input NaNs.

_nextafterl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns the closest representable neighbor of the first argument in the direction toward the second argument.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_nextafter		+		
_nextafterl		+		

1.1.2.14.11 _scalb, _scalbl

Header File

float.h

Category

Math Routines

Prototype

```
double _scalb(double d, long exp);
long double _scalbl(long double ld, long exp);
```

Description

Scales the argument d by a power of 2.

_scalbl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns an exponential value if successful. On overflow (depending on the sign of the argument), the function returns +/–HUGE_VAL; the ERRNO variable is set to ERANGE.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_scalb		+		
_scalbl		+		

1.1.2.14.12 _status87, _statusfp

Header File

float.h

Category

Math Routines

Prototype

```
unsigned int _status87(void);
unsigned int _statusfp(void);
```

Description

Gets floating-point status.

_status87 gets the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

_statusfp is identical to _status87 and is for Microsoft compatibility.

Return Value

The bits in the return value give the floating-point status. See float.h for a complete definition of the bits returned by _status87 and _status87.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_status87		+		
_statusfp		+		

1.1.2.14.13 CW_DEFAULT #define

Header File

float.h

Description

Default control word for 80x87 math coprocessor.

1.1.2.15 io.h

The following functions, macros, and classes are provided in `io.h`:

1.1.2.15.1 access, _waccess

Header File

io.h

Category

Input/output Routines

Prototype

```
int access(const char *filename, int amode);
int _waccess(const wchar_t *filename, int amode);
```

Description

Determines accessibility of a file.

access checks the file named by filename to determine if it exists, and whether it can be read, written to, or executed.

The list of amode values is as follows:

06	Check for read and write permission
04	Check for read permission
02	Check for write permission
01	Execute (ignored)
00	Check for existence of file

All existing files have read access (amode equals 04), so 00 and 04 give the same result. Similarly, amode values of 06 and 02 are equivalent because under Win32 write access implies read access.

If filename refers to a directory, access simply determines whether the directory exists.

Return Value

If the requested access is allowed, access returns 0; otherwise, it returns a value of -1, and the global variable errno is set to one of the following values:

ENOENT	Path or file name not found
EACCES	Permission denied

Example

```
 #include <stdio.h>
 #include <io.h>
 int file_exists(char *filename);
 int main(void)
 {
printf("Does NOTEXIST.FIL exist: %s\n",
 file_exists("NOTEXISTS.FIL") ? "YES" : "NO");
return 0;
 }
 int file_exists(char *filename)
 {
   return (access(filename, 0) == 0);
 }
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
access	+	+		
_waccess		NT only		

1.1.2.15.2 chmod, _wchmod

Header File

io.h

Category

Input/output Routines

Prototype

```
int chmod(const char *path, int amode);
int _wchmod(const wchar_t *path, int amode);
```

Description

Changes file access mode.

chmod sets the file-access permissions of the file given by path according to the mask given by amode. path points to a string.

amode can contain one or both of the symbolic constants S_IWRITE and S_IREAD (defined in sys\stat.h).

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD | S_IWRITE	Permission to read and write (write permission implies read permission)

Return Value

Upon successfully changing the file access mode, chmod returns 0. Otherwise, chmod returns a value of -1.

In the event of an error, the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Example

```
#include <errno.h>
#include <stdio.h>
#include <io.h>
#include <process.h>
#include <sys\stat.h>
void main(void)
{
  char filename[64];
  struct stat stbuf;
  int amode;
  printf("Enter name of file: ");
  scanf("%s", filename);
  if (stat(filename, &stbuf) != 0)
  {
    perror("Unable to get file information");
    exit(1);
  }
  if (stbuf.st_mode & S_IWRITE)
  {
    printf("Changing to read-only\n");
    amode = S_IREAD;
  }
  else
  {
    printf("Changing to read-write\n");
    amode = S_IREAD|S_IWRITE;
  }
  if (chmod(filename, amode) != 0)
  {
    perror("Unable to change file mode");
    exit(1);
  }
  exit(0);
```

```
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
chmod	+	+		
_wchmod		NT only		

1.1.2.15.3 chsize

Header File

io.h

Category

Input/output Routines

Prototype

```
int chsize(int handle, long size);
```

Description

Changes the file size.

chsize changes the size of the file associated with handle. It can truncate or extend the file, depending on the value of size compared to the file's original size.

The mode in which you open the file must allow writing.

If chsize extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

Return Value

On success, chsize returns 0. On failure, it returns -1 and the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number
ENOSPC	No space left on device

Example

```
#include <string.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* create text file containing 10 bytes */
   handle = open("DUMMY.FIL", O_CREAT);
   write(handle, buf, strlen(buf));
   /* truncate the file to 5 bytes in size */
   chsize(handle, 5);
   /* close the file */
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.4 clearerr

Header File

stdio.h

Category

Input/output Routines

Prototype

```
void clearerr(FILE *stream);
```

Description

Resets error indication.

clearerr resets the named stream's error and end-of-file indicators to 0. Once the error indicator is set, stream operations continue to return error status until a call is made to clearerr or rewind. The end-of-file indicator is reset with each input operation.

Return Value

None.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *fp;
   char ch;
   /* open a file for writing */
   fp = fopen("DUMMY.FIL", "w");
   /* force an error condition by attempting to read */
   ch = fgetc(fp);
   printf("%c\n",ch);
   if (ferror(fp))
   {
      /* display an error message */
      printf("Error reading from DUMMY.FIL\n");
      /* reset the error and EOF indicators */
      clearerr(fp);
   }
   fclose(fp);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.5 close

Header File

io.h

Category

Input/output Routines

Prototype

```
int close(int handle);
```

Description

Closes a file.

The close function closes the file associated with handle, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open.

It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one.

Return Value

Upon successful completion, close returns 0.

On error (if it fails because handle is not the handle of a valid, open file), close returns a value of -1 and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
main()
{
  int handle;
  char buf[11] = "0123456789";
  /* create a file containing 10 bytes */
  handle = open("NEW.FIL", O_CREAT);
  if (handle > -1)
  {
    write(handle, buf, strlen(buf));
    close(handle);                    /* close the file */
  }
  else
  {
    printf("Error opening file\n");
  }
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.6 _creat, _wcreat

Header File

io.h

Category

Input/output Routines

Prototype

```
int creat(const char *path, int amode);
```

```
int _wcreat(const wchar_t *path, int amode);
```

Description

Creates a new file or overwrites an existing one.

Note: Remember that a backslash in a path requires '\\'.

creat creates a new file or prepares to rewrite an existing file given by path. amode applies only to newly created files.

A file created with creat is always created in the translation mode specified by the global variable _fmode (O_TEXT or O_BINARY).

If the file exists and the write attribute is set, creat truncates the file to a length of 0 bytes, leaving the file attributes unchanged. If the existing file has the read-only attribute set, the creat call fails and the file remains unchanged.

The creat call examines only the S_IWRITE bit of the access-mode word amode. If that bit is 1, the file can be written to. If the bit is 0, the file is marked as read-only. All other operating system attributes are set to 0.

amode can be one of the following (defined in sys\stat.h):

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD / S_IWRITE	Permission to read and write (write permission implies read permission)

Return Value

Upon successful completion, _creat returns the new file handle, a nonnegative integer; otherwise, it returns -1.

In the event of error, the global variable errno is set to one of the following:

EACCES	Permission denied
ENOENT	Path or file name not found
EMFILE	Too many open files

Example

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* change the default file mode from text to binary */
```

```
    _fmode = O_BINARY;
    /* create a binary file for reading and writing */
    handle = creat("DUMMY.FIL", S_IREAD |S_IWRITE);
    /* write 10 bytes to the file */
    write(handle, buf, strlen(buf));
    /* close the file */
    close(handle);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
creat	+	+		
_wcreat		NT only		

1.1.2.15.7 creatnew

Header File

io.h

Category

Input/output Routines

Prototype

```
int creatnew(const char *path, int mode);
```

Description

Creates a new file.

creatnew is identical to _rtl_creat with one exception: If the file exists, creatnew returns an error and leaves the file untouched.

The mode argument to creatnew can be zero or an OR-combination of any one of the following constants (defined in dos.h):

FA_HIDDEN	Hidden file
FA_RDONLY	Read-only attribute
FA_SYSTEM	System file

Return Value

Upon successful completion, creatnew returns the new file handle, a nonnegative integer; otherwise, it returns -1.

In the event of error, the global variable errno is set to one of the following values:

EACCES	Permission denied
EEXIST	File already exists
EMFILE	Too many open files
ENOENT	Path or file name not found

Example

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
```

```
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* attempt to create a file that doesn't already exist */
   handle = creatnew("DUMMY.FIL", 0);
if (handle == -1)
      printf("DUMMY.FIL already exists.\n");
   else
   {
      printf("DUMMY.FIL successfully created.\n");
      write(handle, buf, strlen(buf));
      close(handle);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.8 creattemp

Header File

io.h

Category

Input/output Routines

Prototype

```
int creattemp(char *path, int attrib);
```

Description

Creates a unique file in the directory associated with the path name.

A file created with creattemp is always created in the translation mode specified by the global variable _fmode (O_TEXT or O_BINARY).

path is a path name ending with a backslash (\). A unique file name is selected in the directory given by path. The newly created file name is stored in the path string supplied. path should be long enough to hold the resulting file name. The file is not automatically deleted when the program terminates.

The attrib argument to creattemp can be zero or an OR-combination of any one of the following constants (defined in dos.h):

FA_HIDDEN	Hidden file
FA_RDONLY	Read-only attribute
FA_SYSTEM	System file

Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

Return Value

Upon successful completion, the new file handle, a nonnegative integer, is returned; otherwise, -1 is returned.

In the event of error, the global variable errno is set to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
int main(void)
{
   int handle;
   char pathname[128];
   strcpy(pathname, "\\");
   /* create a unique file in the root directory */
   handle = creattemp(pathname, 0);
   printf("%s was the unique file created.\n", pathname);
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.9 dup

Header File

io.h

Category

Input/output Routines

Prototype

```
int dup(int handle);
```

Description

Duplicates a file handle.

- dup creates a new file handle that has the following in common with the original file handle:
- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

handle is a file handle obtained from a call to creat, open, dup, dup2, _rtl_creat, or _rtl_open.

Return Value

Upon successful completion, dup returns the new file handle, a nonnegative integer; otherwise, dup returns -1.

In the event of error, the global variable errno is set to one of the following values:

EBADF	Bad file number
EMFILE	Too many open files

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
void flush(FILE *stream);
int main(void)
{
  FILE *fp;
  char msg[] = "This is a test";
  /* create a file */
  fp = fopen("DUMMY.FIL", "w");
  /* write some data to the file */
  fwrite(msg, strlen(msg), 1, fp);
  printf("Press ENTER to flush DUMMY.FIL:");
  getchar();
  /* flush the data to DUMMY.FIL without closing it */
  flush(fp);
  printf("\nFile was flushed, Press ENTER to quit:");
  getchar();
  return 0;
}
void flush(FILE *stream)
{
  int duphandle;
  /* flush TC's internal buffer */
  fflush(stream);
  /* make a duplicate file handle */
  duphandle = dup(fileno(stream));
  /* close the duplicate handle to flush the DOS buffer */
  close(duphandle);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.10 dup2

Header File

io.h

Category

Input/output Routines

Prototype

```
int dup2(int oldhandle, int newhandle);
```

Description

Duplicates a file handle (oldhandle) onto an existing file handle (newhandle).

- dup2 creates a new file handle that has the following in common with the original file handle:

- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

dup2 creates a new handle with the value of newhandle. If the file associated with newhandle is open when dup2 is called, the file is closed.

newhandle and oldhandle are file handles obtained from a creat, open, dup, or dup2 call.

Return Value

dup2 returns 0 on successful completion, -1 otherwise.

In the event of error, the global variable errno is set to one of the following values:

EBADF Bad file number

EMFILE Too many open files

Example

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>
#define STDOUT 1
int main(void)
{
   int nul, oldstdout;
   char msg[] = "This is a test";
   /* create a file */
   nul = open("DUMMY.FIL", O_CREAT | O_RDWR,
      S_IREAD | S_IWRITE);
   /* create a duplicate handle for standard output */
   oldstdout = dup(STDOUT);
   /*
      redirect standard output to DUMMY.FIL
      by duplicating the file handle onto
      the file handle for standard output.
   */
   dup2(nul, STDOUT);
   /* close the handle for DUMMY.FIL */
   close(nul);
   /* will be redirected into DUMMY.FIL */
   write(STDOUT, msg, strlen(msg));
   /* restore original standard output handle */
   dup2(oldstdout, STDOUT);
   /* close duplicate handle for STDOUT */
   close(oldstdout);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.11 eof

Header File

io.h

Category

Input/output Routines

Prototype

```
int eof(int handle);
```

Description

Checks for end-of-file.

eof determines whether the file associated with handle has reached end-of-file.

Return Value

If the current position is end-of-file, eof returns the value 1; otherwise, it returns 0. A return value of -1 indicates an error; the global variable errno is set to

EBADF	Bad file number

Example

```
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "This is a test";
   char ch;
   /* create a file */
   handle = open("DUMMY.FIL",
                 O_CREAT | O_RDWR,
                 S_IREAD | S_IWRITE);
   /* write some data to the file */
   write(handle, msg, strlen(msg));
   /* seek to the beginning of the file */
   lseek(handle, 0L, SEEK_SET);
   /* reads chars from the file until it reaches EOF */
   do
   {
      read(handle, &ch, 1);
      printf("%c", ch);
   } while (!eof(handle));
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.12 fclose

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fclose(FILE *stream);
```

Description

Closes a stream.

fclose closes the named stream. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with setbuf or setvbuf are not automatically freed. (But if setvbuf is passed null for the buffer pointer it will free it upon close.)

Return Value

fclose returns 0 on success. It returns EOF if any errors were detected.

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.13 _fcloseall

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _fcloseall(void);
```

Description

Closes open streams.

_fcloseall closes all open streams except

stdauxstdstreams

When _fcloseall flushes the associated buffers before closing a stream. The buffers allocated by the system are released.

Note: stdprn and stdaux streams are not available in Win32.

Return Value

_fcloseall returns the total number of streams it closed. The _fcloseall function returns EOF if any errors were detected.

Example

```
#include <stdio.h>
int main(void)
{
   int streams_closed;
   /* open two streams */
   fopen("DUMMY.ONE", "w");
   fopen("DUMMY.TWO", "w");
   /* close the open streams */
```

```
   streams_closed = fcloseall();
   if (streams_closed == EOF)
      /* issue an error message */
      perror("Error");
   else
      /* print result of fcloseall() function */
      printf("%d streams were closed.\n", streams_closed);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.14 _fdopen, _wfdopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *_fdopen(int handle, char *type);
```

```
FILE *_wfdopen(int handle, wchar_t *type);
```

Description

Associates a stream with a file handle.

_fdopen associates a stream with a file handle obtained from creat, dup, dup2, or open.

The type of stream must match the mode of the open handle.

The type string used in a call to _fdopen is one of the following values:

r	Open for reading only. _fdopen returns NULL if the file cannot be opened.
w	Create for writing. If the file already exists, its contents are overwritten.
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing). _fdopen returns NULL if the file cannot be opened.
w+	Create a new file for update. If the file already exists, its contents are overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode, append t to the value of the type string (for example, rt or w+t).

Similarly, to specify binary mode, append b to the type string (for example, rb or w+b).

If t or b is not given in the type string, the mode is governed by the global variable _fmode.

If _fmode is set to O_BINARY, files will be opened in binary mode.

If _fmode is set to O_TEXT, files will be opened in text mode.

Note: The O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

Return Value

On successful completion _fdopen returns a pointer to the newly opened stream. In the event of error it returns NULL.

Example

```
#include <sys\stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   FILE *stream;
   /* open a file */
   handle = open("DUMMY.FIL", O_CREAT,
                 S_IREAD | S_IWRITE);
   /* now turn the handle into a stream */
   stream = fdopen(handle, "w");
   if (stream == NULL)
      printf("fdopen failed\n");
   else
   {
      fprintf(stream, "Hello world\n");
      fclose(stream);
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fdopen	+	+		
_wfdopen		+		

1.1.2.15.15 feof

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int feof(FILE *stream);
```

Description

Detects end-of-file on a stream.

feof is a macro that tests the given stream for an end-of-file indicator. Once the indicator is set read operations on the file return the indicator until rewind is called or the file is closed. The end-of-file indicator is reset with each input operation.

Return Value

feof returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file has not been reached.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   /* open a file for reading */
   stream = fopen("DUMMY.FIL", "r");
   /* read a character from the file */
   fgetc(stream);
   /* check for EOF */
   if (feof(stream))
      printf("We have reached end-of-file\n");
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.16 ferror

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int ferror(FILE *stream);
```

Description

Detects errors on stream.

ferror is a macro that tests the given stream for a read or write error. If the stream's error indicator has been set it remains set until clearerr or rewind is called or until the stream is closed.

Return Value

ferror returns nonzero if an error was detected on the named stream.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   /* open a file for writing */
   stream = fopen("DUMMY.FIL", "w");
   /* force an error condition by attempting to read */
   (void) getc(stream);
   if (ferror(stream))  /* test for an error on the stream */
```

```
    {
       /* display an error message */
       printf("Error reading from DUMMY.FIL\n");
       /* reset the error and EOF indicators */
       clearerr(stream);
    }
    fclose(stream);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.17 fflush

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fflush(FILE *stream);
```

Description

Flushes a stream.

If the given stream has buffered output fflush writes the output for stream to the associated file.

The stream remains open after fflush has executed. fflush has no effect on an unbuffered stream.

Return Value

fflush returns 0 on success. It returns EOF if any errors were detected.

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
void flush(FILE *stream);
int main(void)
{
   FILE *stream;
   char msg[] = "This is a test";
   /* create a file */
   stream = fopen("DUMMY.FIL", "w");
   /* write some data to the file */
   fwrite(msg, strlen(msg), 1, stream);
   printf("Press ENTER to flush DUMMY.FIL:");
   getchar();
   /* flush the data to DUMMY.FIL without closing it */
   flush(stream);
   printf("\nFile was flushed, Press ENTER to quit:");
   getchar();
   return 0;
}
void flush(FILE *stream)
```

```
{
     int duphandle;
     /* flush the stream's internal buffer */
     fflush(stream);
     /* make a duplicate file handle */
     duphandle = dup(fileno(stream));
     /* close the duplicate handle to flush the DOS buffer */
     close(duphandle);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.18 fgetc, fgetwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fgetc(FILE *stream);
```

```
wint_t fgetwc(FILE *stream);
```

Description

Gets character from stream.

fgetc returns the next character on the named input stream.

Return Value

On success fgetc returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char string[] = "This is a test";
   char ch;
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* write a string into the file */
   fwrite(string, strlen(string), 1, stream);
   /* seek to the beginning of the file */
   fseek(stream, 0, SEEK_SET);
   do
   {
      /* read a char from the file */
      ch = fgetc(stream);
      /* display the character */
      putchar(ch);
   } while (ch != EOF);
```

```
   fclose(stream);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fgetc	+	+	+	+
fgetwc		+	+	+

1.1.2.15.19 _fgetchar, _fgetwchar

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int _fgetchar(void);
```

```
wint_t _fgetwchar(void);
```

Description

Reads a character from stdin.

_fgetchar returns the next character from stdin. It is defined as fgetc(stdin).

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success _fgetchar returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char ch;
   /* prompt the user for input */
   printf("Enter a character followed by <Enter>: ");
   /* read the character from stdin */
   ch = fgetchar();
   /* display what was read */
   printf("The character read is: '%c'\n", ch);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fgetchar		+		
_fgetwchar		+		

1.1.2.15.20 fgetpos

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

Gets the current file pointer.

fgetpos stores the position of the file pointer associated with the given stream in the location pointed to by pos. The exact value is unimportant; its value is opaque except as a parameter to subsequent fsetpos calls.

Return Value

On success fgetpos returns 0. On failure it returns a nonzero value and sets the global variable errno to

EBADF	Bad file number
EINVAL	Invalid number

Example

```
#include <stdlib.h>
#include <stdio.h>
void showpos(FILE *stream);
int main(void)
{
   FILE *stream;
   fpos_t filepos;
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* save the file pointer position */
   fgetpos(stream, &filepos);
   /* write some data to the file */
   fprintf(stream, "This is a test");
   /* show the current file position */
   showpos(stream);
   /* set a new file position, display it */
   if (fsetpos(stream, &filepos) == 0)
     showpos(stream);
   else
   {
      fprintf(stderr, "Error setting file pointer.\n");
      exit(1);
   }
   /* close the file */
   fclose(stream);
   return 0;
}
void showpos(FILE *stream)
{
   fpos_t pos;
   /* display the current file pointer
      position of a stream */
   fgetpos(stream, &pos);
```

```
   printf("File position: %ld\n", pos);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.21 fgets, fgetws

Header File

stdio.h

Category

Input/output Routines

Prototype

```
char *fgets(char *s, int n, FILE *stream);
```

```
wchar_t *fgetws(wchar_t *s, int n, FILE *stream); // Unicode version
```

Description

Gets a string from a stream.

fgets reads characters from stream into the string s. The function stops reading when it reads either n - 1 characters or a newline character whichever comes first. fgets retains the newline character at the end of s. A null byte is appended to s to mark the end of the string.

Return Value

On success fgets returns the string pointed to by s; it returns NULL on end-of-file or error.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char string[] = "This is a test";
   char msg[20];
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* write a string into the file */
   fwrite(string, strlen(string), 1, stream);
   /* seek to the start of the file */
   fseek(stream, 0, SEEK_SET);
   /* read a string from the file */
   fgets(msg, strlen(string)+1, stream);
   /* display the string */
   printf("%s", msg);
   fclose(stream);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fgets	+	+	+	+

fgetws		+	+	+

1.1.2.15.22 filelength

Header File

io.h

Category

Input/output Routines

Prototype

```
long filelength(int handle);
```

Description

Gets file size in bytes.

filelength returns the length (in bytes) of the file associated with handle.

Return Value

On success filelength returns a **long** value the file length in bytes. On error it returns -1 and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* create a file containing 10 bytes */
   handle = open("DUMMY.FIL", O_CREAT);
   write(handle, buf, strlen(buf));
   /* display the size of the file */
   printf("file length in bytes: %ld\n", filelength(handle));
   /* close the file */
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.23 _fileno

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _fileno(FILE *stream);
```

Description

Gets the file handle.

_fileno is a macro that returns the file handle for the given stream. If stream has more than one handle _fileno returns the handle assigned to the stream when it was first opened.

Return Value

_fileno returns the integer file handle associated with stream.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   int handle;
   /* create a file */
   stream = fopen("DUMMY.FIL", "w");
   /* obtain the file handle associated with the stream */
   handle = fileno(stream);
   /* display the handle number */
   printf("handle number: %d\n", handle);
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.24 _findclose

Header File

io.h

Category

Directory Control Routines

Prototype

```
int _findclose(long handle);
```

Description

Closes the specified search handle and releases associated resources associated with previous calls to findfirst/findnext. The handle parameter is the search handle returned by the previous call to _findfirst.

This function is provided for Microsoft compatibility.

Return Value

On success, returns 0.

Otherwise, returns –1 and sets errno to:

ENOENTFile specification that could not be matched

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findclose		+		

See Also

_findfirst (see page 341)

_findnext (see page 342)

1.1.2.15.25 _findfirst, __wfindfirst

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findfirst(char *filter, struct _finddata_t *fileinfo);
long __wfindfirst(wchar_t *filter, struct _wfinddata_t *fileinfo);
```

Description

Begins a search of a disk directory to find information about the first instance of a filename that matches a specified filter. The filter parameter is a string that specifies which files to return. Wildcards may be used in the filter. The fileinfo parameter is the file information buffer. If a matching file is found, the fileinfo structure is filled with the file-directory information.

These functions are provided for Microsoft compatibility.

Return Value

On success, returns a unique search handle to a file or group of files matching the filter specification.

Otherwise, returns –1 and sets errno to one of the following values:

ENOENTPath or file name not found

EINVALInvalid filename specification

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findfirst		+		
__wfindfirst		NT only		

See Also

_findclose (see page 340)

_findnext (see page 342)

1.1.2.15.26 _findfirsti64, _wfindfirsti64

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findfirsti64(char *filter, struct _finddatai64_t *fileinfo);
long _wfindfirsti64(wchar_t *filter, struct _wfinddatai64_t *fileinfo);
```

Description

Begins a search of a disk directory to find information about the first instance of a filename that matches a specified filter. The filter parameter is a string that specifies which files to return. Wildcards may be used in the filter. The fileinfo parameter is the file information buffer. If a matching file is found, the fileinfo structure is filled with the file-directory information.

These i64 versions are for 64 bit filesize use and are provided for Microsoft compatibility.

Return Value

On success, returns a unique search handle identifying the file or group of files matching the filter specification.

Otherwise, returns –1 and sets errno to one of the following values:

ENOENTFile specification that could not be matched

EINVALInvalid filename specification

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findfirsti64		+		
_wfindfirsti64		NT only		

1.1.2.15.27 _findnext, __wfindnext

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findnext(long handle, struct _finddata_t *fileinfo);
long __wfindnext(long handle, struct _wfinddata_t *fileinfo);
```

Description

Finds subsequent files, if any, that match the filter argument in a previous call to _findfirst/__wfindfirst. Then, _findnext/__wfindnext updates the fileinfo structure with the necessary information for continuing the search. One file name for each call to _tfindnext is returned until no more files are found in the directory matching the pathname (filter).

The handle parameter is the search handle returned by a previous call to _findfirst. The fileinfo parameter is the file information buffer.

These functions are provided for Microsoft compatibility.

Return Value

On success, returns 0.

Otherwise, returns –1 and sets errno to:

ENOENTFile specification that could not be matched

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findnext		+		
__wfindnext		NT only		

See Also

_findclose (see page 340)

_findfirst (see page 341)

1.1.2.15.28 _findnexti64, _wfindnexti64

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findnexti64(long handle, struct _finddatai64_t *fileinfo);
__int64 _wfindnexti64(long handle, struct _wfinddata_t *fileinfo);
```

Description

Finds subsequent files, if any, that match the filter argument in a previous call to _findfirsti64/_wfindfirsti64. Then, _findnexti64/_wfindnexti64 updates the fileinfo structure with the necessary information for continuing the search. One file name for each call to _tfindnext is returned until no more files are found in the directory matching the pathname (filter).

The handle parameter is the search handle returned by a previous call to _findfirst. The fileinfo parameter is the file information buffer.

These i64 versions are for 64 bit filesize use and are provided for Microsoft compatibility.

Return Value

On success, returns 0.

Otherwise, returns –1 and sets errno to:

ENOENTFile specification that could not be matched

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findnexti64		+		
_wfindnexti64		NT only		

1.1.2.15.29 _flushall

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _flushall(void);
```

Description

Flushes all streams.

_flushall clears all buffers associated with open input streams and writes all buffers associated with open output streams to their respective files. Any read operation following _flushall reads new data into the buffers from the input files. Streams stay open after _flushall executes.

Return Value

_flushall returns an integer the number of open input and output streams.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   /* create a file */
   stream = fopen("DUMMY.FIL", "w");
   /* flush all open streams */
   printf("%d streams were flushed.\n", flushall());
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.30 fopen, _wfopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *fopen(const char *filename, const char *mode);
FILE *_wfopen(const wchar_t *filename, const wchar_t *mode);
```

Description

Opens a stream.

fopen opens the file named by filename and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations.

The mode string used in calls to fopen is one of the following values:

r	Open for reading only.
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a t to the mode string (rt w+t and so on). Similarly to specify binary mode append a b to the mode string (wb a+b and so on). fopen also allows the t or b to be inserted between the letter and the + character in the mode string; for example rt+ is equivalent to r+t.

If a t or b is not given in the mode string the mode is governed by the global variable _fmode. If _fmode is set to O_B/INARY files are opened in binary mode. If _fmode is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

Return Value

On successful completion fopen returns a pointer to the newly opened stream. In the event of error it returns NULL.

Portability

	POSIX	Win32	ANSI C	ANSI C++
fopen	+	+	+	+
_wfopen		NT only		

1.1.2.15.31 fprintf, fwprintf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
int fwprintf(FILE *stream, const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to a stream.

fprintf accepts a series of arguments applies to each a format specifier contained in the format string pointed to by format and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

Note: For details on format specifiers, see printf Format Specifiers.

Return Value

fprintf returns the number of bytes output. In the event of error it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   int i = 100;
   char c = 'C';
   float f = 1.234;
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* write some data to the file */
   fprintf(stream, "%d %c %f", i, c, f);
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fprintf	+	+	+	+
fwprintf		+	+	+

1.1.2.15.32 fputc, fputwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fputc(int c, FILE *stream);
wint_t fputwc(wint_t c, FILE *stream);
```

Description

Puts a character on a stream.

fputc outputs character c to the named stream.

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success, fputc returns the character c. On error, it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char msg[] = "Hello world";
   int i = 0;
   while (msg[i])
   {
      fputc(msg[i], stdout);
      i++;
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fputc	+	+	+	+
fputwc		+	+	+

1.1.2.15.33 _fputchar, _fputwchar

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _fputchar(int c);
```

```
wint_t _fputwchar(wint_t c);
```

Description

Outputs a character to stdout.

_fputchar outputs character c to stdout. _fputchar(c) is the same as fputc(cstdout).

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On success _fputchar returns the character c.

On error it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
```

```
    char msg[] = "This is a test";
    int i = 0;
    while (msg[i])
    {
       fputchar(msg[i]);
       i++;
    }
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fputchar		+		
_fputwchar		+		

1.1.2.15.34 fputs, fputws

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fputs(const char *s, FILE *stream);
```

```
int fputws(const wchar_t *s, FILE *stream);
```

Description

Outputs a string on a stream.

fputs copies the null-terminated string s to the given output stream; it does not append a newline character and the terminating null character is not copied.

Return Value

On success fputs returns a non-negative value.

On error it returns a value of EOF.

Example

```
#include <stdio.h>
int main(void)
{
   /* write a string to standard output */
   fputs("Hello world\n", stdout);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fputs	+	+	+	+
fputws		+	+	+

1.1.2.15.35 fread

Header File

stdio.h

Category

Input/output Routines

Prototype

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Description

Reads data from a stream.

fread reads n items of data each of length size bytes from the given input stream into a block pointed to by ptr.

The total number of bytes read is (n * size).

Return Value

On success fread returns the number of items (not bytes) actually read.

On end-of-file or error it returns a short count (possibly 0).

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char msg[] = "this is a test";
   char buf[20];
   if ((stream = fopen("DUMMY.FIL", "w+"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   /* write some data to the file */
   fwrite(msg, strlen(msg)+1, 1, stream);
   /* seek to the beginning of the file */
   fseek(stream, SEEK_SET, 0);
   /* read the data and display it */
   fread(buf, strlen(msg)+1, 1, stream);
   printf("%s\n", buf);
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.36 freopen, _wfreopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

```
FILE *_wfreopen(const wchar_t *filename, const wchar_t *mode, FILE *stream);
```

Description

Associates a new file with an open stream.

freopen substitutes the named file in place of the open stream. It closes stream regardless of whether the open succeeds. freopen is useful for changing the file attached to stdin, stdout, or stderr.

The mode string used in calls to fopen is one of the following values:

r	Open for reading only.
w	Create for writing. .
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing).
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a t to the mode string (rt w+t and so on); similarly to specify binary mode append a b to the mode string (wb a+b and so on).

If a t or b is not given in the mode string the mode is governed by the global variable _fmode. If _fmode is set to O_BINARY files are opened in binary mode. If _fmode is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

Return Value

On successful completion freopen returns the argument stream.

On error it returns NULL.

Example

```
#include <stdio.h>
int main(void)
{
   /* redirect standard output to a file */
   if (freopen("OUTPUT.FIL", "w", stdout)
       == NULL)
      fprintf(stderr, "error redirecting stdout\n");
   /* this output will go to a file */
   printf("This will go into a file.");
   /* close the standard output stream */
   fclose(stdout);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
freopen	+	+	+	+
_wfreopen		NT only		

1.1.2.15.37 fscanf, fwscanf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fscanf(FILE *stream, const char *format[, address, ...]);
```

```
int fwscanf(FILE *stream, const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from a stream.

fscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to fscanf in the format string pointed to by format. Finally fscanf stores the formatted input at an address passed to it as an argument following format. The number of format specifiers and addresses must be the same as the number of input fields.

Note: For details on format specifiers, see scanf Format Specifiers.

fscanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace) or it can terminate entirely for a number of reasons. See scanf for a discussion of possible causes.

Return Value

fscanf returns the number of input fields successfully scanned, converted and stored. The return value does not include scanned fields that were not stored.

If fscanf attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int i;
   printf("Input an integer: ");
   /* read an integer from the
      standard input stream */
   if (fscanf(stdin, "%d", &i))
      printf("The integer read was: %i\n", i);
   else
   {
      fprintf(stderr, "Error reading an integer from stdin.\n");
      exit(1);
   }
   return 0;
```

```
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.38 fseek

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fseek(FILE *stream, long offset, int whence);
```

Description

Repositions a file pointer on a stream.

fseek sets the file pointer associated with stream to a new position that is offset bytes from the file location given by whence. For text mode streams offset should be 0 or a value returned by ftell.

whence must be one of the values 0. 1, or 2 which represent three symbolic constants (defined in stdio.h) as follows:

fseek discards any character pushed back using ungetc. fseek is used with stream I/O; for file handle I/O use lseek.

After fseek the next operation on an update file can be either input or output.

Return Value

fseek returns 0 if the pointer is successfully moved and nonzero on failure.

fseek might return a 0 indicating that the pointer has been moved successfully when in fact it has not been. This is because DOS, which actually resets the pointer, does not verify the setting. fseek returns an error code only on an unopened file or device.

In the event of an error return the global variable errno is set to one of the following values:

EBADF	Bad file pointer
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

Example

```
#include <stdio.h>
long filesize(FILE *stream);
int main(void)
{
   FILE *stream;
   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "This is a test");
   printf("Filesize of MYFILE.TXT is %ld bytes\n", filesize(stream));
   fclose(stream);
   return 0;
}
long filesize(FILE *stream)
{
```

```
    long curpos, length;
    curpos = ftell(stream);
    fseek(stream, 0L, SEEK_END);
    length = ftell(stream);
    fseek(stream, curpos, SEEK_SET);
    return length;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.39 fsetpos

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

Positions the file pointer of a stream.

fsetpos sets the file pointer associated with stream to a new position. The new position is the value obtained by a previous call to fgetpos on that stream. It also clears the end-of-file indicator on the file that stream points to and undoes any effects of ungetc on that file. After a call to fsetpos the next operation on the file can be input or output.

Return Value

On success fsetpos returns 0.

On failure it returns a nonzero value and also sets the global variable errno to a nonzero value.

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.40 _fsopen, _wfsopen

Header File

stdio.h, share.h

Category

Input/output Routines

Prototype

```
FILE *_fsopen(const char *filename, const char *mode, int shflag);
```

```
FILE *_wfsopen(const wchar_t *filename, const wchar_t *mode, int shflag);
```

Description

Opens a stream with file sharing.

_fsopen opens the file named by filename and associates a stream with it. _fsopen returns a pointer that is used to identify the stream in subsequent operations.

The mode string used in calls to _fsopen is one of the following values:

r	**Open for reading only.**
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end of file. or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a t to the mode string (rt w+t and so on). Similarly to specify binary mode append a b to the mode string (wb a+b and so on). _fsopen also allows the t or b to be inserted between the letter and the + character in the mode string; for example rt+ is equivalent to r+t. If a t or b is not given in the mode string the mode is governed by the global variable _fmode. If _fmode is set to O_BINARY files are opened in binary mode. If _fmode is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream, however:

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

shflag specifies the type of file-sharing allowed on the file filename. Symbolic constants for shflag are defined in share.h.

SH_COMPAT	Sets compatibility mode
SH_DENYRW	Denies read/write access
SH_DENYWR	Denies write access
SH_DENYRD	Denies read access
SH_DENYNONE	Permits read/write access
SH_DENYNO	Permits read/write access

Return Value

On successful completion _fsopen returns a pointer to the newly opened stream.

On error it returns NULL.

Example

```
#include <io.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
int main(void)
{
  FILE *f;
  int status;
  f = _fsopen("TESTFILE.DAT", "r", SH_DENYNO);
  if (f == NULL)
  {
```

```
    printf("_fsopen failed\n");
    exit(1);
  }
  status = access("TESTFILE.DAT", 6);
  if (status == 0)
    printf("read/write access allowed\n");
  else
    printf("read/write access not allowed\n");
  fclose(f);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fsopen		+		
_wfsopen		NT only		

1.1.2.15.41 ftell

Header File

stdio.h

Category

Input/output Routines

Prototype

```
long int ftell(FILE *stream);
```

Description

Returns the current file pointer.

ftell returns the current file pointer for stream. The offset is measured in bytes from the beginning of the file (if the file is binary). The value returned by ftell can be used in a subsequent call to fseek.

Return Value

ftell returns the current file pointer position on success. It returns -1L on error and sets the global variable errno to a positive value.

In the event of an error return the global variable errno is set to one of the following values:

EBADF	Bad file pointer
ESPIPE	Illegal seek on device

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "This is a test");
   printf("The file pointer is at byte %ld\n", ftell(stream));
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.42 fwrite

Header File

stdio.h

Category

Input/output Routines

Prototype

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

Description

Writes to a stream.

fwrite appends n items of data each of length size bytes to the given output file. The data written begins at ptr. The total number of bytes written is (n x size). ptr in the declarations is a pointer to any object.

Return Value

On successful completion fwrite returns the number of items (not bytes) actually written.

On error it returns a short count.

Example

```
#include <stdio.h>
struct mystruct
{
  int i;
  char ch;
};
int main(void)
{
   FILE *stream;
   struct mystruct s;
   if ((stream = fopen("TEST.$$$", "wb")) == NULL) /* open file TEST.$$$ */
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   s.i = 0;
   s.ch = 'A';
   fwrite(&s, sizeof(s), 1, stream); /* write struct s to file */
   fclose(stream); /* close file */
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.43 getc, getwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int getc(FILE *stream);
```

```
wint_t getwc(FILE *stream);
```

Description

Gets character from stream.

getc returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success, getc returns the character read, after converting it to an **int** without sign extension.

On end-of-file or error, it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char ch;
   printf("Input a character:");
/* read a character from the
standard input stream */
   ch = getc(stdin);
   printf("The character input was: '%c'\n", ch);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getc	+	+	+	+
getwc		+	+	+

1.1.2.15.44 getchar, getwchar

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int getchar(void);
wint_t getwchar(void);
```

Description

Gets character from stdin.

getchar is a macro that returns the next character on the named input stream stdin. It is defined to be getc(stdin).

Note: Do not use this function in Win32 GUI applications.

Return Value

On success, getchar returns the character read, after converting it to an **int** without sign extension.

On end-of-file or error, it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   int c;
/*
Note that getchar reads from stdin and is line buffered; this means it will not return until
you press ENTER.
 */
   while ((c = getchar()) != '\n')
      printf("%c", c);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getchar	+	+	+	+
getwchar		+	+	+

1.1.2.15.45 getftime, setftime

Header File

io.h

Category

Input/output Routines

Prototype

```
int getftime(int handle, struct ftime *ftimep);
int setftime(int handle, struct ftime *ftimep);
```

Description

Gets and sets the file date and time.

getftime retrieves the file time and date for the disk file associated with the open handle. The ftime structure pointed to by ftimep is filled in with the file's time and date.

setftime sets the file date and time of the disk file associated with the open handle to the date and time in the ftime structure

pointed to by ftimep. The file must not be written to after the setftime call or the changed information will be lost. The file must be open for writing; an EACCES error will occur if the file is open for read-only access.

setftime requires the file to be open for writing; an EACCES error will occur if the file is open for read-only access.

The ftime structure is defined as follows:

```
struct ftime {
unsigned ft_tsec: 5; /* two seconds */
unsigned ft_min: 6; /* minutes */
unsigned ft_hour: 5; /* hours */
unsigned ft_day: 5; /* days */
unsigned ft_month: 4; /* months */
unsigned ft_year: 7; /* year - 1980*/
};
```

Return Value

getftime and setftime return 0 on success.

In the event of an error return -1 is returned and the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number
EINVFNC	Invalid function number

Example

```
#include <stdio.h>
#include <io.h>
int main(void)
{
   FILE *stream;
   std::ftime ft;
   if ((stream = fopen("TEST.$$$",
        "wt")) == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   getftime(fileno(stream), &ft);
   printf("File time: %u:%u:%u\n",
          ft.ft_hour, ft.ft_min,
          ft.ft_tsec * 2);
   printf("File date: %u/%u/%u\n",
          ft.ft_month, ft.ft_day,
          ft.ft_year+1980);
   fclose(stream);
   return 0;
}
```

1.1.2.15.46 _get_osfhandle

Header File

io.h

Category

Input/output Routines

Prototype

```
long _get_osfhandle(int filehandle);
```

Description

Associates file handles.

The _get_osfhandle function associates an operating system file handle with an existing runtime file handle. The variable filehandle is the file handle of your program.

Return value

On success, _get_osfhandle returns an operating system file handle corresponding to the variable filehandle.

On error, the function returns -1 and sets the global variable errno to

EBADF	an invalid file handle

Example

```
#include <windows.h>
#include <fcntl.h>
#include <stdio.h>
#include <io.h>
//Example for _get_osfhandle() and _open_osfhandle()
BOOL InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam);
Example_get_osfhandle(HWND hWnd);
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;      // message
  if (!InitApplication(hInstance))  // Initialize shared things
   return (FALSE); // Exits if unable to initialize
  /* Perform initializations that apply to a specific instance */
  if (!(InitInstance(hInstance, nCmdShow)))
   return (FALSE);
  /* Acquire and dispatch messages until a WM_QUIT message is received. */
  while (GetMessage(&msg, // message structure
   NULL, // handle of window receiving the message
   NULL, // lowest message to examine
   NULL))  // highest message to examine
  {
  TranslateMessage(&msg); // Translates virtual key codes
  DispatchMessage(&msg);  // Dispatches message to window
  }
  return (msg.wParam);  // Returns the value from PostQuitMessage
}
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;
  // Fill in window class structure with parameters that describe the
  // main window.
  wc.style = CS_HREDRAW | CS_VREDRAW; // Class style(s).
  wc.lpfnWndProc = (long (FAR PASCAL*)(void *,unsigned int,unsigned int, long ))MainWndProc;
// Function to retrieve messages for
                // windows of this class.
  wc.cbClsExtra = 0;  // No per-class extra data.
  wc.cbWndExtra = 0;  // No per-window extra data.
  wc.hInstance = hInstance; // Application that owns the class.
```

```
  wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = NULL;  // Name of menu resource in .RC file.
  wc.lpszClassName = "Example"; // Name used in call to CreateWindow.
  /* Register the window class and return success/failure code. */
  return (RegisterClass(&wc));
}
HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND  hWnd; // Main window handle.
  /* Create a main window for this application instance.  */
  hWnd = CreateWindow(
   "Example",                  // See RegisterClass() call.
   "Example _get_osfhandle _open_osfhandle (32 bit)",  // Text for window title bar.
   WS_OVERLAPPEDWINDOW,        // Window style.
   CW_USEDEFAULT,         // Default horizontal position.
   CW_USEDEFAULT,          // Default vertical position.
   CW_USEDEFAULT,            // Default width.
   CW_USEDEFAULT,        // Default height.
   NULL,                  // Overlapped windows have no parent.
   NULL,                  // Use the window class menu.
   hInstance,              // This instance owns this window.
   NULL                   // Pointer not needed.
  );
  /* If window could not be created, return "failure" */
  if (!hWnd)
   return (FALSE);
  /* Make the window visible; update its client area; and return "success" */
  ShowWindow(hWnd, nCmdShow); // Show the window
  UpdateWindow(hWnd);     // Sends WM_PAINT message
  return (hWnd);        // Returns the value from PostQuitMessage
}
#pragma argsused
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{
  switch (message)
  {
    case WM_CREATE:
    {
      Example_get_osfhandle(hWnd);
      return NULL;
    }
    case WM_QUIT:
    case WM_DESTROY:  // message: window being destroyed
      PostQuitMessage(0);
    break;
    default:      // Passes it on if unprocessed
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
}
Example_get_osfhandle(HWND hWnd)
{
    long osfHandle;
    char str[128];
    int fHandle = open("file1.c", O_CREAT|O_TEXT);
    if(fHandle != -1)
    {
      osfHandle = _get_osfhandle(fHandle);
      sprintf(str, "file handle = %lx OS file handle = %lx", fHandle, osfHandle);
      MessageBox(hWnd,str,"_get_osfhandle",MB_OK|MB_ICONINFORMATION);
      close(fHandle);
      fHandle = _open_osfhandle(osfHandle, O_TEXT );
      sprintf(str, "file handle = %lx OS file handle = %lx", fHandle, osfHandle);
      MessageBox(hWnd,str,"_open_osfhandle",MB_OK|MB_ICONINFORMATION);
```

```
        close(fHandle);
    }
    else
      MessageBox(hWnd,"File Open Error","WARNING",MB_OK|MB_ICONSTOP);
return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.47 gets, _getws

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
char *gets(char *s);
```

```
wchar_t *_getws(wchar_t *s); // Unicode version
```

Description

Gets a string from stdin.

gets collects a string of characters terminated by a new line from the standard input stream stdin and puts it into s. The new line is replaced by a null character (\0) in s.

gets allows input strings to contain certain whitespace characters (spaces, tabs). gets returns when it encounters a new line; everything up to the new line is copied into s.

The gets function is not length-terminated. If the input string is sufficiently large, data can be overwritten and corrupted. The fgets function provides better control of input strings.

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success, gets returns the string argument s.

On end-of-file or error, it returns NULL

Example

```
#include <stdio.h>
int main(void)
{
   char string[80];
   printf("Input a string:");
   gets(string);
   printf("The string input was: %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
gets	+	+	+	+
_getws		+		

1.1.2.15.48 _getw

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _getw(FILE *stream);
```

Description

Gets an integer from stream.

_getw returns the next integer in the named input stream. It assumes no special alignment in the file.

_getw should not be used when the stream is opened in text mode.

Return Value

_getw returns the next integer on the input stream.

On end-of-file or error, _getw returns EOF.

Note: Because EOF is a legitimate value for _getw to return, feof or ferror should be used to detect end-of-file or error.

Example

```
#include <stdio.h>
#include <stdlib.h>
#define FNAME "test.$$$"
int main(void)
{
FILE *fp;
int word;
/* place the word in a file */
fp = fopen(FNAME, "wb");
if (fp == NULL)
{
printf("Error opening file %s\n", FNAME);
      exit(1);
   }
   word = 94;
   putw(word,fp);
   if (ferror(fp))
       printf("Error writing to file\n");
   else
       printf("Successful write\n");
   fclose(fp);
   /* reopen the file */
   fp = fopen(FNAME, "rb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
```

```
      exit(1);
   }
   /* extract the word */
   word = getw(fp);
   if (ferror(fp))
       printf("Error reading file\n");
   else
       printf("Successful read: word = %d\n", word);
   /* clean up */
   fclose(fp);
   unlink(FNAME);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.49 isatty

Header File

io.h

Category

Input/output Routines

Prototype

```
int isatty(int handle);
```

Description

Checks for device type.

isatty determines whether handle is associated with any one of the following character devices:

- a terminal
- a console
- a printer
- a serial port

Return Value

If the device is one of the four character devices listed above, isatty returns a nonzero integer. If it is not such a device, isatty returns 0.

Example

```
#include <stdio.h>
#include <io.h>
int main(void)
{
    int handle;
    handle = fileno(stdout);
    if (isatty(handle))
       printf("Handle %d is a device type\n", handle);
    else
       printf("Handle %d isn't a device type\n", handle);
    return 0;
```

```
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.50 lock

Header File

io.h

Category

Input/output Routines

Prototype

```
int lock(int handle, long offset, long length);
```

Description

Sets file-sharing locks.

lock provides an interface to the operating system file-sharing mechanism.

A lock can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

Return Value

lock returns 0 on success. On error, lock returns -1 and sets the global variable errno to

EACCES	Locking violation

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
int main(void)
{
   int handle, status;
   long length;
   handle = _sopen("c:\\autoexec.bat",
      O_RDONLY,SH_DENYNO,S_IREAD);
   if (handle < 0)
   {
      printf("_sopen failed\n");
      exit(1);
   }
   length = filelength(handle);
   status = lock(handle,0L,length/2);
   if (status == 0)
      printf("lock succeeded\n");
   else
      printf("lock failed\n");
   status = unlock(handle,0L,length/2);
   if (status == 0)
```

```
      printf("unlock succeeded\n");
   else
      printf("unlock failed\n");
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.51 locking

Header File

io.h, sys\locking.h

Category

Input/output Routines

Prototype

```
int locking(int handle, int cmd, long length);
```

Description

Sets or resets file-sharing locks.

locking provides an interface to the operating system file-sharing mechanism. The file to be locked or unlocked is the open file specified by handle. The region to be locked or unlocked starts at the current file position, and is length bytes long.

Locks can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

The cmd specifies the action to be taken (the values are defined in sys\locking.h):

LK_LOCK	Lock the region. If the lock is unsuccessful, try once a second for 10 seconds before giving up.
LK_RLCK	Same as LK_LOCK.
LK_NBLCK	Lock the region. If the lock if unsuccessful, give up immediately.
LK_NBRLCK	Same as LK_NBLCK.
LK_UNLCK	Unlock the region, which must have been previously locked.

Return Value

On successful operations, locking returns 0. Otherwise, it returns -1, and the global variable errno is set to one of the following values:

EACCES	File already locked or unlocked
EBADF	Bad file number
EDEADLOCK	File cannot be locked after 10 retries (cmd is LK_LOCK or LK_RLCK)
EINVAL	Invalid cmd, or SHARE.EXE not loaded

Example

```
#include <io.h>
```

```
#include <fcntl.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
#include <sys\locking.h>
int main(void)
{
  int handle, status;
  long length;
  handle = sopen("c:\\autoexec.bat", O_RDONLY,SH_DENYNO);
  if (handle < 0) {
    printf("sopen failed\n");
    exit(1);
  }
  length = filelength(handle);
  status = locking(handle,LK_LOCK,length/2);
  if (status == 0)
    printf("lock succeeded\n");
  else
    perror("lock failed");
  status = locking(handle,LK_UNLCK,length/2);
  if (status == 0)
    printf("unlock succeeded\n");
  else
    perror("unlock failed");
  close(handle);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.52 lseek

Header File

io.h

Category

Input/output Routines

Prototype

```
long lseek(int handle, long offset, int fromwhere);
```

Description

Moves file pointer.

lseek sets the file pointer associated with handle to a new position offset bytes beyond the file location given by fromwhere. fromwhere must be one of the following symbolic constants (defined in io.h):

SEEK_CUR	Current file pointer position
SEEK_END	End-of-file
SEEK_SET	File beginning

Return Value

lseek returns the offset of the pointer's new position measured in bytes from the file beginning. lseek returns -1L on error, and the global variable errno is set to one of the following values:

EBADF	Bad file handle
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

Example

```
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
    int handle;
    char msg[] = "This is a test";
    char ch;
    /* create a file */
    handle = open("TEST.$$$", O_CREAT | O_RDWR, S_IREAD | S_IWRITE);
    /* write some data to the file */
    write(handle, msg, strlen(msg));
    /* seek to the beginning of the file */
    lseek(handle, 0L, SEEK_SET);
    /* reads chars from the file until we hit EOF */
    do
    {
       read(handle, &ch, 1);
       printf("%c", ch);
    }  while (!eof(handle));
    close(handle);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.53 _open_osfhandle

Header File

io.h

Category

Input/output Routines

Prototype

```
int _open_osfhandle(long osfhandle, int flags);
```

Description

Associates file handles.

The _open_osfhandle function allocates a runtime file handle and sets it to point to the operating system file handle specified by

osfhandle.

The value flags is a bitwise OR combination of one or more of the following manifest constants (defined in fcntl.h):

O_APPEND	Repositions the file pointer to the end of the file before every write operation.
O_RDONLY	Opens the file for reading only.
O_TEXT	Opens the file in text (translated) mode.

Return Value

On success, _open_osfhandle returns a C runtime file handle. Otherwise, it returns -1.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.54 _pclose

Header File

stdio.h

Category

Input/output Routines, Process Control Routines

Prototype

```
int _pclose(FILE * stream);
```

Description

Waits for piped command to complete.

_pclose closes a pipe stream created by a previous call to _popen, and then waits for the associated child command to complete.

Return Value

On success, _pclose returns the termination status of the child command. This is the same value as the termination status returned by cwait, except that the high and low order bytes of the low word are swapped.

On error, it returns -1.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.55 _popen, _wpopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *_popen (const char *command, const char *mode);
```

```
FILE *_wpopen (const wchar_t *command, const wchar_t *mode);
```

Description

Creates a command processor pipe.

The _popen function creates a pipe to the command processor. The command processor is executed asynchronously, and is passed the command line in command. The mode string specifies whether the pipe is connected to the command processor's standard input or output, and whether the pipe is to be opened in binary or text mode.

The mode string can take one of the following values:

rt	Read child command's standard output (text).
rb	Read child command's standard output (binary).
wt	Write to child command's standard input (text).
wb	Write to child command's standard input (binary).

The terminating t or b is optional; if missing, the translation mode is determined by the external variable _fmode.

Use the _pclose function to close the pipe and obtain the return code of the command.

Return Value

On success, _popen returns a FILE pointer that can be used to read the standard output of the command, or to write to the standard input of the command, depending on the mode string.

On error, it returns NULL.

Example

```
/* this program initiates a child process to run the dir command
   and pipes the directory listing from the child to the parent.
*/
#include <stdio.h>     // popen() pclose() feof() fgets() puts()
#include <string.h>    // strlen()
int main( )
{
  FILE* handle;        // handle to one end of pipe
  char message[256];   // buffer for text passed through pipe
  int status;          // function return value
  // open a pipe to receive text from a process running "DIR"
  handle = _popen("dir /b", "rt");
  if (handle == NULL)
  {
    perror("_popen error");
  }
  // read and display input received from the child process
  while (fgets(message, sizeof(message), handle))
  {
    fprintf(stdout, message);
  }
  // close the pipe and check the return status
    status = _pclose(handle);
  if (status == -1)
  {
    perror("_pclose error");
  }
  return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_popen		+		
_wpopen		NT only		

1.1.2.15.56 printf, wprintf

Header File

stdio.h

Category

Console I/O Routines

Prototype

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

```
int wprintf(const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to stdout.

The printf function:

- Accepts a series of arguments
- Applies to each argument a format specifier contained in the format string *format
- Outputs the formatted data (to the screen, a stream, stdout, or a string)

There must be enough arguments for the format. If there are not, the results will be unpredictable and likely disastrous. Excess arguments (more than required by the format) are merely ignored.

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On success, printf returns the number of bytes output.

On error, printf returns EOF.

More About printf

Example

```
#include <stdio.h>
#include <string.h>
#define I 555
#define R 5.5
int main(void)
{
   int i,j,k,l;
   char buf[7];
   char *prefix = buf;
   char tp[20];
   printf("prefix  6d      6o      8x        10.2e        "
          "10.2f\n");
   strcpy(prefix,"%");
   for (i = 0; i < 2; i++)
```

```
    {
       for (j = 0; j < 2; j++)
          for (k = 0; k < 2; k++)
             for (l = 0; l < 2; l++)
             {
                if (i==0)  strcat(prefix,"-");
                if (j==0)  strcat(prefix,"+");
                if (k==0)  strcat(prefix,"#");
                if (l==0)  strcat(prefix,"0");
                printf("%5s |",prefix);
                strcpy(tp,prefix);
                strcat(tp,"6d |");
                printf(tp,I);
                strcpy(tp,"");
                strcpy(tp,prefix);
                strcat(tp,"6o |");
                printf(tp,I);
                strcpy(tp,"");
                strcpy(tp,prefix);
                strcat(tp,"8x |");
                printf(tp,I);
                strcpy(tp,"");
                strcpy(tp,prefix);
                strcat(tp,"10.2e |");
                printf(tp,R);
                strcpy(tp,prefix);
                strcat(tp,"10.2f |");
                printf(tp,R);
                printf("  \n");
                strcpy(prefix,"%");
             }
        }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
printf	+	+	+	+
_wprintf		+		

1.1.2.15.57 putc, putwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int putc(int c, FILE *stream);
```

```
wint_t putwc(wint_t c, FILE *stream);
```

Description

Outputs a character to a stream.

putc is a macro that outputs the character c to the stream given by stream.

Return Value

On success, putc returns the character printed, c.

On error, putc returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char msg[] = "Hello world\n";
   int i = 0;
   while (msg[i])
      putc(msg[i++], stdout);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
putc	+	+	+	+
putwc		+	+	+

1.1.2.15.58 putchar, putwchar

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int putchar(int c);
```

```
wint_t putwchar(wint_t c);
```

Description

putchar is a macro defined to be putc(c, stdout).

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On success, putchar returns the character c. On error, putchar returns EOF.

Example

```
#include <stdio.h>
/* define some box-drawing characters */
#define LEFT_TOP  0xDA
#define RIGHT_TOP 0xBF
#define HORIZ     0xC4
#define VERT      0xB3
#define LEFT_BOT  0xC0
#define RIGHT_BOT 0xD9
int main(void)
{
   char i, j;
```

```
   /* draw the top of the box */
   putchar(LEFT_TOP);
   for (i=0; i<10; i++)
      putchar(HORIZ);
   putchar(RIGHT_TOP);
   putchar('\n');
   /* draw the middle */
   for (i=0; i<4; i++)
   {
      putchar(VERT);
      for (j=0; j<10; j++)
         putchar(' ');
      putchar(VERT);
      putchar('\n');
   }
   /* draw the bottom */
   putchar(LEFT_BOT);
   for (i=0; i<10; i++)
      putchar(HORIZ);
   putchar(RIGHT_BOT);
   putchar('\n');
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
putchar	+	+	+	+
putwchar		+	+	+

1.1.2.15.59 puts, _putws

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int puts(const char *s);
```

```
int _putws(const wchar_t *s);
```

Description

Outputs a string to stdout.

puts copies the null-terminated string s to the standard output stream stdout and appends a newline character.

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On successful completion, puts returns a nonnegative value. Otherwise, it returns a value of EOF.

Example

```
#include <stdio.h>
int main(void)
{
```

```
   char string[] = "This is an example output string\n";
   puts(string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
puts	+	+	+	+
_putws		+		

1.1.2.15.60 _putw

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _putw(int w, FILE *stream);
```

Description

Writes an integer on a stream.

_putw outputs the integer w to the given stream. _putw neither expects nor causes special alignment in the file.

Return Value

On success, _putw returns the integer w. On error, _putw returns EOF. Because EOF is a legitimate integer, use ferror to detect errors with _putw.

Example

```
#include <stdio.h>
#include <stdlib.h>
#define FNAME "test.$$$"
int main(void)
{
   FILE *fp;
   int word;
   /* place the word in a file */
   fp = fopen(FNAME, "wb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }
   word = 94;
   putw(word,fp);
   if (ferror(fp))
       printf("Error writing to file\n");
   else
       printf("Successful write\n");
   fclose(fp);
   /* reopen the file */
   fp = fopen(FNAME, "rb");
   if (fp == NULL)
   {
```

```
        printf("Error opening file %s\n", FNAME);
        exit(1);
    }
    /* extract the word */
    word = getw(fp);
    if (ferror(fp))
        printf("Error reading file\n");
    else
        printf("Successful read: word = %d\n", word);
    /* clean up */
    fclose(fp);
    unlink(FNAME);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.61 read

Header File

io.h

Category

Input/output Routines

Prototype

```
int read(int handle, void *buf, unsigned len);
```

Description

Reads from file.

read attempts to read len bytes from the file associated with handle into the buffer pointed to by buf.

For a file opened in text mode, read removes carriage returns and reports end-of-file when it reaches a Ctrl-Z.

The file handle handle is obtained from a creat, open, dup, or dup2 call.

On disk files, read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that read can read is UINT_MAX -1, because UINT_MAX is the same as -1, the error return indicator. UINT_MAX is defined in limits.h.

Return Value

On successful completion, read returns an integer indicating the number of bytes placed in the buffer. If the file was opened in text mode, read does not count carriage returns or Ctrl-Z characters in the number of bytes read.

On end-of-file, read returns 0. On error, read returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
   void *buf;
   int handle, bytes;
   buf = malloc(10);
/*
Looks for a file in the current directory named TEST.$$$ and attempts to read 10 bytes from
it. To use this example you should create the file TEST.$$$.
 */
   if ((handle =
      open("TEST.$$$", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }
   if ((bytes = read(handle, buf, 10)) == -1) {
      printf("Read Failed.\n");
      exit(1);
   }
   else {
      printf("Read: %d bytes read.\n", bytes);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.62 remove, _wremove

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int remove(const char *filename);
```

```
int _wremove(const wchar_t *filename);
```

Description

Removes a file.

remove deletes the file specified by filename. It is a macro that simply translates its call to a call to unlink. If your file is open, be sure to close it before removing it.

The filename string can include a full path.

Return Value

On successful completion, remove returns 0. On error, it returns -1, and the global variable errno is set to one of the following

values:

EACCES	Permission denied
ENOENT	No such file or directory

Example

```
#include <stdio.h>
int main(void)
{
    char file[80];
    /* prompt for file name to delete */
    printf("File to delete: ");
    gets(file);
    /* delete the file */
    if (remove(file) == 0)
       printf("Removed %s.\n",file);
    else
       perror("remove");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
remove	+	+	+	+
_wremove		NT only		

1.1.2.15.63 rename, _wrename

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int rename(const char *oldname, const char *newname);
```

```
int _wrename(const wchar_t *oldname, const wchar_t *newname);
```

Description

Renames a file.

rename changes the name of a file from oldname to newname. If a drive specifier is given in newname, the specifier must be the same as that given in oldname.

Directories in oldname and newname need not be the same, so rename can be used to move a file from one directory to another. Wildcards are not allowed.

This function will fail (EEXIST) if either file is currently open in any process.

Return Value

On success, rename returns 0.

On error (if the file cannot be renamed), it returns -1 and the global variable errno is set to one of the following values:

EEXIST	Permission denied: file already exists.
ENOENT	No such file or directory
ENOTSAM	Not same device

Example

```
#include <stdio.h>
int main(void)
{
    char oldname[80], newname[80];
    /* prompt for file to rename and new name */
    printf("File to rename: ");
    gets(oldname);
    printf("New name: ");
    gets(newname);
    /* Rename the file */
    if (rename(oldname, newname) == 0)
       printf("Renamed %s to %s.\n", oldname, newname);
    else
       perror("rename");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
rename	+	+	+	+
_wrename		NT only		

1.1.2.15.64 rewind

Header File

stdio.h

Category

Input/output Routines

Prototype

```
void rewind(FILE *stream);
```

Description

Repositions a file pointer to the beginning of a stream.

rewind(stream) is equivalent to fseek(stream, 0L, SEEK_SET), except that rewind clears the end-of-file and error indicators, while fseek clears the end-of-file indicator only.

After rewind, the next operation on an update file can be either input or output.

Return Value

None.

Example

```
#include <stdio.h>
#include <dir.h>
```

```
int main(void)
{
    FILE *fp;
    char *fname = "TXXXXXX", *newname, first;
    newname = mktemp(fname);
    fp = fopen(newname,"w+");
    fprintf(fp,"abcdefghijklmnopqrstuvwxyz");
    rewind(fp);
    fscanf(fp,"%c",&first);
    printf("The first character is: %c\n",first);
    fclose(fp);
    remove(newname);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.65 _rmtmp

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _rmtmp(void);
```

Description

Removes temporary files.

The _rmtmp function closes and deletes all open temporary file streams which were previously created with tmpfile. The current directory must the same as when the files were created, or the files will not be deleted.

Return Value

_rmtmp returns the total number of temporary files it closed and deleted.

Example

```
#include <stdio.h>
#include <process.h>

void main()
{
   FILE *stream;
   int i;

   /* Create temporary files */
   for (i = 1; i <= 10; i++)
   {
     if ((stream = tmpfile()) == NULL)
       perror("Could not open temporary file\n");
     else
       printf("Temporary file %d created\n", i);
   }
   /* Remove temporary files */
```

```
   if (stream != NULL)
     printf("%d temporary files deleted\n", rmtmp());
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.66 _rtl_chmod, _wrtl_chmod

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_chmod(const char *path, int func [, int attrib]);
```

```
int _wrtl_chmod(const wchar_t *path, int func, ... );
```

Description

Gets or sets file attributes.

Note: The _rtl_chmod function replaces _chmod which is obsolete

_rtl_chmod can either fetch or set file attributes. If func is 0, _rtl_chmod returns the current attributes for the file. If func is 1, the attribute is set to attrib.

attrib can be one of the following symbolic constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

Return Value

On success, _rtl_chmod returns the file attribute word.

On error, it returns a value of -1 and sets the global variable errno to one of the following values:

ENOENT	Path or filename not found
EACCES	Permission denied

Example

```
#include <errno.h>
#include <stdio.h>
#include <dir.h>
```

```
#include <io.h>
int get_file_attrib(char *filename);
int main(void)
{
  char filename[128];
  int attrib;
  printf("Enter a filename:");
  scanf("%s", filename);
  attrib = get_file_attrib(filename);
  if (attrib == -1)
     switch(errno)
     {
        case ENOENT : printf("Path or file not found.\n");
                      break;
        case EACCES : printf("Permission denied.\n");
                      break;
        default:      printf("Error number: %d", errno);
                      break;
     }
  else
     {
        if (attrib & FA_RDONLY)
           printf("%s is read-only.\n", filename);
        if (attrib & FA_HIDDEN)
           printf("%s is hidden.\n", filename);
        if (attrib & FA_SYSTEM)
           printf("%s is a system file.\n", filename);
        if (attrib & FA_DIREC)
           printf("%s is a directory.\n", filename);
        if (attrib & FA_ARCH)
           printf("%s is an archive file.\n", filename);
     }
  return 0;
}
/* returns the attributes of a DOS file */
int get_file_attrib(char *filename)
{
   return(_rtl_chmod(filename, 0));
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rtl_chmod		+		
_wrtl_chmod		NT only		

1.1.2.15.67 _rtl_close

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_close(int handle);
```

Description

Closes a file.

Note: This function replaces _close which is obsolete

The _rtl_close function closes the file associated with handle, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open.

It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one.

Return Value

On success, _rtl_close returns 0.

On error (if it fails because handle is not the handle of a valid, open file), _rtl_close returns a value of -1 and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = _rtl_open("TEST.$$$", O_RDWR)) == -1)
   {
      perror("Error:");
      return 1;
   }
   _rtl_write(handle, msg, strlen(msg));
   _rtl_close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.68 _rtl_creat, _wrtl_creat

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_creat(const char *path, int attrib);
```

```
int _wrtl_creat(const wchar_t *path, int attrib);
```

Description

Creates a new file or overwrites an existing one.

Note: The _rtl_creat function replaces _creat, which is obsolete

_rtl_creat opens the file specified by path. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

The attrib argument is an OR'ed combination of one or more of the following constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return Value

On success, _rtl_creat returns the new file handle (a non-negative integer).

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCESPermission denied

EMFILEToo many open files

ENOENTPath or file name not found

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
int main() {
  unsigned count;
  int handle;
  char buf[11] = "0123456789";
  /* Create a 10-byte file using _rtl_creat. */
  if ((handle = _rtl_creat("DUMMY2.FIL", 0)) < 0) {
    perror("Unable to _rtl_create DUMMY2.FIL");
    return 1;
  }
  if (_rtl_write(handle, buf, strlen(buf)) < 0) {
    perror("Unable to _rtl_write to DUMMY2.FIL");
    return 1;
  }
  _rtl_close(handle);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rtl_crea		+		
_wrtl_creat		NT only		

1.1.2.15.69 _rtl_open, _wrtl_open

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_open(const char *filename, int oflags);
int _wrtl_open(const wchar_t *path, int oflags);
```

Description

Opens a file for reading or writing.

Note: The _rtl_open function replaces _open which is obsolete.

_rtl_open opens the file specified by filename, then prepares it for reading or writing, as determined by the value of oflags. The file is always opened in binary mode.

oflags uses the flags from the following two lists. Only one flag from List 1 can be used (and one must be used) and the flags in List 2 can be used in any logical combination.

O_RDONLY	Open for reading.
O_WRONLY	Open for writing.
O_RDWR	Open for reading and writing.

The following additional values can be included in oflags (using an OR operation):

O_NOINHERIT	The file is not passed to child programs.
SH_COMPAT	Allow other opens with SH_COMPAT. All other openings of a file with the SH_COMPAT flag must be opened using SH_COMPAT flag. You can request a file open that uses SH_COMPAT logically OR'ed with some other flag (for example, SH_COMPAT | SH_DENWR is allowed). The call will fail if the file has already been opened in any other shared mode.
SH_DENYRW	Only the current handle can have access to the file.
SH_DENWR	Allow only reads from any other open to the file.
SH_DENYRD	Allow only writes from any other open to the file.
SH_DENYNO	Allow other shared opens to the file, but not other SH_COMPAT opens.

Note: These symbolic constants are defined in fcntl.h and share.h.

Only one of the SH_DENYxx values can be included in a single _rtl_open routine. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is defined by HANDLE_MAX.

Return Value

On success:_rtl_open returns a non-negative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files

ENOENT	Path or file not found

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = _rtl_open("TEST.$$$", O_RDWR)) == -1)
   {
      perror("Error:");
      return 1;
   }
   _rtl_write(handle, msg, strlen(msg));
   _rtl_close(handle);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rtl_open		+		
_wrtl_open		NT only		

1.1.2.15.70 _rtl_read

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_read(int handle, void *buf, unsigned len);
```

Description

Reads from file.

Note: This function replaces _read which is obsolete.

This function reads len bytes from the file associated with handle into the buffer pointed to by buf. When a file is opened in text mode, _rtl_read does not remove carriage returns.

The argument handle is a file handle obtained from a creat, open, dup, or dup2 call.

On disk files, _rtl_read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes it can read is UINT_MAX -1 (because UINT_MAX is the same as -1, the error return indicator). UINT_MAX is defined in limits.h.

Return Value

On success, _rtl_read returns either

- a positive integer, indicating the number of bytes placed in the buffer
- zero, indicating end-of-file

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
   void *buf;
   int handle, bytes;
   buf = malloc(10);
/*
Looks for a file in the current directory named TEST.$$$ and attempts to read 10 bytes from
it. To use this example you should create the file TEST.$$$
 */
   if ((handle =
       open("TEST.$$$", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }
   if ((bytes = _rtl_read(handle, buf, 10)) == -1) {
      printf("Read Failed.\n");
      exit(1);
   }
   else {
      printf("_rtl_read: %d bytes read.\n", bytes);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.71 _rtl_write

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_write(int handle, void *buf, unsigned len);
```

Description

Writes to a file.

Note: This function replaces _write which is obsolete.

_rtl_write attempts to write len bytes from the buffer pointed to by buf to the file associated with handle.

The maximum number of bytes that _rtl_write can write is UINT_MAX -1 (because UINT_MAX is the same as -1), which is the error return indicator for _rtl_write. UINT_MAX is defined in limits.h. _rtl_write does not translate a linefeed character (LF) to a CR/LF pair because all its files are binary files.

If the number of bytes actually written is less than that requested the condition should be considered an error and probably indicates a full disk.

For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

For files opened with the O_APPEND option, the file pointer is not positioned to EOF before writing the data.

Return Value

On success, _rtl_write returns number of bytes written.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
   void *buf;
   int handle, bytes;
   buf = malloc(200);
/*
Create a file name TEST.$$$ in the current directory and writes 200 bytes to it. If TEST.$$$
already exists, it's overwritten.
 */
   if ((handle = open("TEST.$$$", O_CREAT | O_WRONLY | O_BINARY,
                      S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }
   if ((bytes = _rtl_write(handle, buf, 200)) == -1) {
      printf("Write Failed.\n");
      exit(1);
   }
   printf("_rtl_write: %d bytes written.\n",bytes);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.72 scanf, wscanf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int scanf(const char *format[, address, ...]);
int wscanf(const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from the stdin stream.

Note: For Win32 GUI applications, stdin must be redirected.

The scanf function:

- scans a series of input fields one character at a time
- formats each field according to a corresponding format specifier passed in the format string *format.
- vsscanf scans and formats input from a string, using an argument list

There must be one format specifier and address for each input field.

scanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely. For details about why this might happen, see When ...scanf Stops Scanning.

Note: scanf often leads to unexpected results if you diverge from an expected pattern. You must provide information that tells scanf how to synchronize at the end of a line.

The combination of gets or fgets followed by sscanf is safe and easy, and therefore recommended over scanf.

Return Value

On success, scanf returns the number of input fields successfully scanned, converted, and stored. The return value does not include scanned fields that were not stored.

On error:

- if no fields were stored, scanf returns 0.
- if scanf attempts to read at end-of-file or at end-of-string, it returns EOF.

More About scanf

Example

```
#include <stdio.h>
int main(void)
{
   char label[20];
   char name[20];
   int entries = 0;
   int loop, age;
   double salary;
   struct Entry_struct
   {
```

```
      char  name[20];
      int   age;
      float salary;
   } entry[20];
/* Input a label as a string of characters restricting to 20 characters */
   printf("\n\nPlease enter a label for the chart: ");
   scanf("%20s", label);
   fflush(stdin);  /* flush the input stream in case of bad input */
/* Input number of entries as an integer */
  printf("How many entries will there be? (less than 20) ");
  scanf("%d", &entries);
  fflush(stdin);   /* flush the input stream in case of bad input */
/* input a name restricting input to only letters upper or lower case */
   for (loop=0;loop<entries;++loop)
   {
      printf("Entry %d\n", loop);
      printf("  Name   : ");
      scanf("%[A-Za-z]", entry[loop].name);
      fflush(stdin);  /* flush the input stream in case of bad input */
/* input an age as an integer */
      printf("  Age    : ");
      scanf("%d", &entry[loop].age);
      fflush(stdin);  /* flush the input stream in case of bad input */
/* input a salary as a float */
      printf("  Salary : ");
      scanf("%f", &entry[loop].salary);
      fflush(stdin); /* flush the input stream in case of bad input */
   }
/* Input a name, age and salary as a string, integer, and double */
   printf("\nPlease enter your name, age and salary\n");
   scanf("%20s %d %lf", name, &age, &salary);
/* Print out the data that was input */
   printf("\n\nTable %s\n",label);
   printf("Compiled by %s  age %d  $%15.2lf\n", name, age, salary);
   printf("-----------------------------------------------------\n");
   for (loop=0;loop<entries;++loop)
      printf("%4d | %-20s | %5d | %15.2lf\n",
         loop + 1,
         entry[loop].name,
         entry[loop].age,
         entry[loop].salary);
   printf("-----------------------------------------------------\n");
   return 0;
}
```

1.1.2.15.73 setbuf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
void setbuf(FILE *stream, char *buf);
```

Description

Assigns buffering to a stream.

setbuf causes the buffer buf to be used for I/O buffering instead of an automatically allocated buffer. It is used after stream has been opened.

If buf is null, I/O will be unbuffered; otherwise, it will be fully buffered. The buffer must be BUFSIZ bytes long (specified in stdio.h).

stdin and stdout are unbuffered if they are not redirected; otherwise, they are fully buffered. setbuf can be used to change the buffering style used.

Unbuffered means that characters written to a stream are immediately output to the file or device, while buffered means that the characters are accumulated and written as a block.

setbuf produces unpredictable results unless it is called immediately after opening stream or after a call to fseek. Calling setbuf after stream has been unbuffered is legal and will not cause problems.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return Value

None.

Example

```
#include <stdio.h>
/* BUFSIZ is defined in stdio.h */
char outbuf[BUFSIZ];
int main(void)
{
   /* attach a buffer to the standard output stream */
   setbuf(stdout, outbuf);
   /* put some characters into the buffer */
   puts("This is a test of buffered output.\n\n");
   puts("This output will go into outbuf\n");
   puts("and won't appear until the buffer\n");
   puts("fills up or we flush the stream.\n");
   /* flush the output buffer */
   fflush(stdout);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.74 setmode

Header File

io.h

Category

Input/output Routines

Prototype

```
int setmode(int handle, int amode);
```

Description

Sets mode of an open file.

setmode sets the mode of the open file associated with handle to either binary or text. The argument amode must have a value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in fcntl.h.)

Return Value

setmode returns the previous translation mode if successful. On error it returns -1 and sets the global variable errno to

EINVAL	Invalid argument

Example

```
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
int main (int argc, char ** argv )
(
  FILE *fp;
  int newmode;
  long where;
  char buf[256];
  fp = fopen( argv[1], "r+" );
  if ( !fp )
  {
    printf( "Couldn't open %s\n", argv[1] );
    return -1;
  }
  newmode = setmode( fileno( fp ), O_BINARY );
  if ( newmode == -1 )
  {
    printf( "Couldn't set mode of %s\n", argv[1] );
    return -2
  }
  fp->flags |= _F_BIN;
  where = ftell( fp );
  printf ("file position: %d\n", where );
  fread( buf, 1, 1, fp );
  where = ftell ( fp );
  printf( "read %c, file position: %ld\n", *buf, where );
  fclose ( fp );
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.75 setvbuf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Description

Assigns buffering to a stream.

setvbuf causes the buffer buf to be used for I/O buffering instead of an automatically allocated buffer. It is used after the given

stream is opened.

If buf is null, a buffer will be allocated using malloc; the buffer will use size as the amount allocated. The buffer will be automatically freed on close. The size parameter specifies the buffer size and must be greater than zero.

The parameter size is limited by the constant UINT_MAX as defined in limits.h.

stdin and stdout are unbuffered if they are not redirected; otherwise, they are fully buffered. Unbuffered means that characters written to a stream are immediately output to the file or device, while buffered means that the characters are accumulated and written as a block.

- The type parameter is one of the following:

_IOFBF	fully buffered file. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file.
_IOLBF	line buffered file. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output, however, the buffer will be flushed whenever a newline character is written to the file.
_IONBF	unbuffered file. The buf and size parameters are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return Value

On success, setvbuf returns 0.

On error (if an invalid value is given for type or size, or if there is not enough space to allocate a buffer), it returns nonzero.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *input, *output;
   char bufr[512];
   input = fopen("file.in", "r+b");
   output = fopen("file.out", "w");
   /* set up input stream for minimal disk access,
      using our own character buffer */
if (setvbuf(input, bufr, _IOFBF, 512) != 0)
      printf("failed to set up buffer for input file\n");
   else
      printf("buffer set up for input file\n");
   /* set up output stream for line buffering using space that
      will be obtained through an indirect call to malloc */
   if (setvbuf(output, NULL, _IOLBF, 132) != 0)
      printf("failed to set up buffer for output file\n");
   else
      printf("buffer set up for output file\n");
   /* perform file I/O here */
   /* close files */
   fclose(input);
   fclose(output);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.76 snprintf;snwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int snprintf(char* buffer, size_t nsize, const char* fmt, ...);
int snwprintf(wchar_t* buffer, size_t nsize, const wchar_t* fmt, ...);
```

Description

Sends formatted output to a buffer of a maximum length specified by nsize.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating '\0' character.
- == nsize, then nsize characters are written, with no terminating '\0' character.

> nsize, then only nsize characters are written, with no terminating '\0' character.

If nsize is 0, then the string will not be written to (and may be NULL).

Return Value

Number of bytes output, or, if nsize is 0, the number of bytes needed, not including the terminating '\0' character.

1.1.2.15.77 _snprintf;_snwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Syntax

```
int _snprintf(char* buffer, size_t nsize, const char* format, ...);
int _snwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, ...);
```

Description

Sends formatted output to a string of a maximum length specified by nsize. _snprintf and _snwprintf are Microsoft compatible with the _snprintf and _snprintfw functions, respectively.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating '\0' character.
- == nsize, then nsize characters are written with no terminating '\0' character.

If nsize is 0, then the string will not be written to (and may be NULL).

If nsize is too small, then return value is -1, and only nsize characters are written, with no terminating '\0' character.

Return Value

Number of bytes output or –1 if nsize is too small.

1.1.2.15.78 sprintf, swprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int sprintf(char *buffer, const char *format[, argument, ...]);
```

```
int swprintf(wchar_t *buffer, const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to a string.

Note: For details on format specifiers, see printf.

sprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string.

sprintf applies the first format specifier to the first argument, the second to the second, and so on. There must be the same number of format specifiers as arguments.

Return Value

On success, sprintf returns the number of bytes output. The return value does not include the terminating null byte in the count.

On error, sprintf returns EOF.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   char buffer[80];
   sprintf(buffer, "An approximation of Pi is %f\n", M_PI);
   puts(buffer);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sprintf	+	+	+	+
swprintf		+	+	+

1.1.2.15.79 sscanf, swscanf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Syntax

```
int sscanf(const char *buffer, const char *format[, address, ...]);
```

```
int swscanf(const wchar_t *buffer, const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from a string.

Note: For details on format specifiers, see scanf.

sscanf scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specifier passed to sscanf in the format string pointed to by format. Finally, sscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields.

sscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

On success, sscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If sscanf attempts to read at end-of-string, it returns EOF.

On error (If no fields were stored), it returns 0.

Example

```
#include <stdio.h>
#include <stdlib.h>
char *names[4] = {"Peter", "Mike", "Shea", "Jerry"};
#define NUMITEMS 4
int main(void)
{
   int   loop;
   char  temp[4][80];
   char  name[20];
   int   age;
   long  salary;
/* create name, age and salary data */
   for (loop=0; loop < NUMITEMS; ++loop)
      sprintf(temp[loop], "%s %d %ld", names[loop], random(10) + 20, random(5000) + 27500L);
/* print title bar */
   printf("%4s | %-20s | %5s | %15s\n", "#", "Name", "Age", "Salary");
   printf("   ---------------------------------------------\n");
/* input a name, age and salary data */
   for (loop=0; loop < NUMITEMS; ++loop)
      {
      sscanf(temp[loop],"%s %d %ld", &name, &age, &salary);
      printf("%4d | %-20s | %5d | %15ld\n", loop + 1, name, age, salary);
```

```
      }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sscanf	+	+	+	+
swscanf		+	+	+

1.1.2.15.80 _strerror

Header File

string.h, stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
char *_strerror(const char *s);
```

Description

Builds a customized error message.

_strerror lets you generate customized error messages; it returns a pointer to a null-terminated string containing an error message.

- If s is null, the return value points to the most recent error message.
- If s is not null, the return value contains s (your customized error message), a colon, a space, the most-recently generated system error message, and a new line. s should be 94 characters or less.

Return Value

_strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to _strerror.

Example

```
#include <stdio.h>
#include <errno.h>
int main(void)
{
   char *buffer;
   buffer = strerror(errno);
   printf("Error: %s\n", buffer);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.81 tell

Header File

io.h

Category

Input/output Routines

Prototype

```
long tell(int handle);
```

Description

Gets the current position of a file pointer.

tell gets the current position of the file pointer associated with handle and expresses it as the number of bytes from the beginning of the file.

Return Value

tell returns the current file pointer position. A return of -1 (**long**) indicates an error, and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = open("TEST.$$$", O_CREAT | O_TEXT | O_APPEND)) == -1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   printf("The file pointer is at byte %ld\n", tell(handle));
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.82 _tempnam, _wtempnam

Header File

stdio.h

Category

Input/output Routines

Prototype

```
char *_tempnam(char *dir, char *prefix)
```

```
wchar_t *_wtempnam(wchar_t *dir, wchar_t *prefix)
```

Description

Creates a unique file name in specified directory.

The _tempnam function accepts single-byte or multibyte string arguments.

The _tempnam function creates a unique file name in arbitrary directories. The unique file is not actually created; _tempnam only verifies that it does not currently exist. It attempts to use the following directories, in the order shown, when creating the file name:

- The directory specified by the TMP environment variable.
- The dir argument to _tempnam.
- The P_tmpdir definition in stdio.h. If you edit stdio.h and change this definition, _tempnam will not use the new definition.
- The current working directory.

If any of these directories is NULL, or undefined, or does not exist, it is skipped.

The prefix argument specifies the first part of the file name; it cannot be longer than 5 characters, and cannot contain a period (.). A unique file name is created by concatenating the directory name, the prefix, and 6 unique characters. Space for the resulting file name is allocated with malloc; when this file name is no longer needed, the caller should call free to free it.

If you do create a temporary file using the name constructed by _tempnam, it is your responsibility to delete the file name (for example, with a call to remove). It is not deleted automatically. (tmpfile does delete the file name.)

Return Value

If _tempnam is successful, it returns a pointer to the unique temporary file name, which the caller can pass to free when it is no longer needed. Otherwise, if _tempnam cannot create a unique file name, it returns NULL.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  FILE *stream;
  int i;
  char *name;
  for (i = 1; i <= 10; i++) {
    if ((name = tempnam("\\tmp","wow")) == NULL)
      perror("tempnam couldn't create name");
    else {
      printf("Creating %s\n",name);
      if ((stream = fopen(name,"wb")) == NULL)
        perror("Could not open temporary file\n");
      else
        fclose(stream);
    }
    free(name);
  }
  printf("Warning: temp files not deleted.\n");
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_tempnam		+		
_wtempnam		+		

1.1.2.15.83 tmpfile

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *tmpfile(void);
```

Description

Opens a “scratch” file in binary mode.

tmpfile creates a temporary binary file and opens it for update (w + b). If you do not change the directory after creating the temporary file, the file is automatically removed when it’s closed or when your program terminates.

Return Value

tmpfile returns a pointer to the stream of the temporary file created. If the file can’t be created, tmpfile returns NULL.

Example

```
#include <stdio.h>
#include <process.h>
int main(void)
{
   FILE *tempfp;
   tempfp = tmpfile();
   if (tempfp)
      printf("Temporary file created\n");
   else
   {
      printf("Unable to create temporary file\n");
      exit(1);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.15.84 tmpnam, _wtmpnam

Header File

stdio.h

Category

Input/output Routines

Prototype

```
char *tmpnam(char *s);
```

```
wchar_t *_wtmpnam(wchar_t *s);
```

Description

Creates a unique file name.

tmpnam creates a unique file name, which can safely be used as the name of a temporary file. tmpnam generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in stdio.h as 65,535.

The parameter to tmpnam, s, is either null or a pointer to an array of at least L_tmpnam characters. L_tmpnam is defined in stdio.h. If s is NULL, tmpnam leaves the generated temporary file name in an internal static object and returns a pointer to that object. If s is not NULL, tmpnam overwrites the internal static object and places its result in the pointed-to array, which must be at least L_tmpnam characters long, and returns s.

If you do create such a temporary file with tmpnam, it is your responsibility to delete the file name (for example, with a call to remove). It is not deleted automatically. (tmpfile does delete the file name.)

Return Value

If s is null, tmpnam returns a pointer to an internal static object. Otherwise, tmpnam returns s.

Example

```
#include <stdio.h>
int main(void)
{
   char name[13];
   tmpnam(name);
   printf("Temporary name: %s\n", name);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tmpnam	+	+	+	+
_wtmpnam		+		

1.1.2.15.85 umask

Header File

io.h, sys\stat.h

Category

Input/output Routines

Prototype

```
unsigned umask(unsigned mode);
```

Description

Sets file read/write permission mask.

The umask function sets the access permission mask used by open and creat. Bits that are set in mode will be cleared in the

access permission of files subsequently created by open and creat.

The mode can have one of the following values, defined in sys\stat.h:

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read and write

Return Value

The previous value of the mask. There is no error return.

Example

```
#include <io.h>
#include <stdio.h>
#include <sys\stat.h>
#define FILENAME "TEST.$$$"
int main(void)
{
    unsigned oldmask;
    FILE *f;
    struct stat statbuf;
    /* Cause subsequent files to be created as read-only */
    oldmask = umask(S_IWRITE);
    printf("Old mask = 0x%x\n",oldmask);
    /* Create a zero-length file */
    if ((f = fopen(FILENAME,"w+")) == NULL)
    {
        perror("Unable to create output file");
        return (1);
    }
    fclose(f);
    /* Verify that the file is read-only */
    if (stat(FILENAME,&statbuf) != 0)
    {
        perror("Unable to get information about output file");
        return (1);
    }
    if (statbuf.st_mode & S_IWRITE)
        printf("Error! %s is writable!\n",FILENAME);
    else
        printf("Success! %s is not writable.\n",FILENAME);
    return (0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.86 ungetc, ungetwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int ungetc(int c, FILE *stream);
```

```
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

Pushes a character back into input stream.

Note: Do not use this function in Win32 GUI applications.

ungetc pushes the character c back onto the named input stream, which must be open for reading. This character will be returned on the next call to getc or fread for that stream. One character can be pushed back in all situations. A second call to ungetc without a call to getc will force the previous character to be forgotten. A call to fflush, fseek, fsetpos, or rewind erases all memory of any pushed-back characters.

Return Value

On success, ungetc returns the character pushed back.

On error, it returns EOF.

Example

```
#include <stdio.h>
#include <ctype.h>
int main( void )
{
   int i=0;
   char ch;
   puts("Input an integer followed by a char:");
   /* read chars until non digit or EOF */
   while((ch = getchar()) != EOF && isdigit(ch))
      i = 10 * i + ch - 48; /* convert ASCII into int value */
   /* if non digit char was read, push it back into input buffer */
   if (ch != EOF)
      ungetc(ch, stdin);
   printf("i = %d, next char in buffer = %c\n", i, getchar());
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ungetc	+	+	+	+
ungetwc		+	+	+

1.1.2.15.87 unlock

Header File

io.h

Category

Input/output Routines

Prototype

```
int unlock(int handle, long offset, long length);
```

Description

Releases file-sharing locks.

unlock provides an interface to the operating system file-sharing mechanism. unlock removes a lock previously placed with a call to lock. To avoid error, all locks must be removed before a file is closed. A program must release all locks before completing.

Return Value

On success, unlock returns 0

O error, it returns -1.

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
int main(void)
{
   int handle, status;
   long length;
   handle = _sopen("c:\\autoexec.bat",O_RDONLY,SH_DENYNO,S_IREAD);
   if (handle < 0)
   {
       printf("_sopen failed\n");
       exit(1);
   }
   length = filelength(handle);
   status = lock(handle,0L,length/2);
   if (status == 0)
      printf("lock succeeded\n");
   else
      printf("lock failed\n");
   status = unlock(handle,0L,length/2);
   if (status == 0)
      printf("unlock succeeded\n");
   else
      printf("unlock failed\n");
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.15.88 vfprintf, vfwprintf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int vfprintf(FILE *stream, const char *format, va_list arglist);
```

```
int vfwprintf(FILE *stream, const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to a stream.

The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Printf Format Specifiers.

vfprintf accepts a pointer to a series of arguments, applies to each argument a format specifier contained in the format string pointed to by format, and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

Return Value

On success, vfprintf returns the number of bytes output.

On error, it returns EOF.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
FILE *fp;
int vfpf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vfprintf(fp, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char string[4] = "abc";
   fp = tmpfile();
   if (fp == NULL)
   {
      perror("tmpfile() call");
      exit(1);
   }
   vfpf("%d %f %s", inumber, fnumber, string);
   rewind(fp);
   fscanf(fp,"%d %f %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   fclose(fp);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
vfprintf	+	+	+	+
vfwprintf		+	+	+

1.1.2.15.89 vfscanf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int vfscanf(FILE *stream, const char *format,va_list arglist);
```

Description

Scans and formats input from a stream.

The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Scanf Format Specifiers.

vfscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to vfscanf in the format string pointed to by format. Finally vfscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields. vfscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character or it might terminate entirely for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vfscanf returns the number of input fields successfully scanned converted and stored; the return value does not include scanned fields that were not stored. If no fields were stored the return value is 0.

If vfscanf attempts to read at end-of-file the return value is EOF.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
FILE *fp;
int vfsf(char *fmt, ...)
{
   va_list  argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vfscanf(fp, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
         char string[4] = "abc";
   fp = tmpfile();
   if (fp == NULL)
   {
      perror("tmpfile() call");
      exit(1);
   }
   fprintf(fp,"%d %f %s\n",inumber,fnumber,string);
   rewind(fp);
   vfsf("%d %f %s",&inumber,&fnumber,string);
   printf("%d %f %s\n",inumber,fnumber,string);
   fclose(fp);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

1.1.2.15.90 vprintf, vwprintf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int vprintf(const char *format, va_list arglist);
int vwprintf(const wchar_t * format, va_list arglist);
```

Description

Writes formatted output to stdout.

Note: Do not use this function in Win32 GUI applications.

The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

Note: For details on format specifiers, see Printf Format Specifiers.

vprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to stdout. There must be the same number of format specifiers as arguments.

Return Value

vprint returns the number of bytes output. In the event of error, vprint returns EOF.

Example

```
#include <stdio.h>
#include <stdarg.h>
int vpf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vprintf(fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char *string = "abc";
   vpf("%d %f %s\n",inumber,fnumber,string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
vprintf	+	+	+	+
vwprintf		+	+	+

1.1.2.15.91 vscanf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int vscanf(const char *format, va_list arglist);
```

Description

Scans and formats input from stdin.

Note: Do not use this function in Win32 GUI applications.

The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

Note: For details on format specifiers, see Scanf Format Specifiers.

vscanf scans a series of input fields, one character at a time, reading from stdin. Then each field is formatted according to a format specifier passed to vscanf in the format string pointed to by format. Finally, vscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields.

vscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If vscanf attempts to read at end-of-file, the return value is EOF.

Example

```
#include <stdio.h>
#include <stdarg.h>
int vscnf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   printf("Enter an integer, a float, and a string (e.g. i,f,s,)\n");
   va_start(argptr, fmt);
   cnt = vscanf(fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
```

```
{
   int inumber;
   float fnumber;
   char string[80];
   vscnf("%d, %f, %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

1.1.2.15.92 vsnprintf;vsnwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int vsnprintf(char* buffer, size_t nsize, const char* format, va_list param);
```

```
int vsnwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, va_list param);
```

Description

Sends formatted output to a buffer of maximum length specified by nsize.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating ‘\0’ character.
- == nsize, then nsize characters are written, with no terminating ‘\0’ character.

> nsize, then only nsize characters are written, with no terminating ‘\0’ character.

If nsize is 0, then the string will not be written to (and may be NULL).

Return Value

Number of bytes output, or, if nsize is 0, the number of bytes needed, not including the terminating ‘\0’ character.

1.1.2.15.93 _vsnprintf;_vsnwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int _vsnprintf(char* buffer, size_t nsize, const char* format, va_list param);
```

```
int _vsnwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, va_list param);
```

Description

Sends formatted output to a string of a maximum length specified by nsize. _vsnprintf and _vsnwprintf are Microsoft compatible with the _vsnprintf and _vsnprintfw functions, respectively.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating '\0' character.
- == nsize, then nsize characters are written with no terminating '\0' character.

If nsize is 0, then the string will not be written to (and may be NULL).

If nsize is too small, then return value is -1, and only nsize characters are written, with no terminating '\0' character.

Return Value

Number of bytes output or –1 if nsize is too small.

1.1.2.15.94 vsprintf, vswprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int vsprintf(char *buffer, const char *format, va_list arglist);
```

```
int vswprintf(wchar_t *buffer, const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to a string.

The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vsprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string. There must be the same number of format specifiers as arguments.

Return Value

vsprintf returns the number of bytes output. In the event of error, vsprintf returns EOF.

Example

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
char buffer[80];
int vspf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vsprintf(buffer, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
```

```
   int inumber = 30;
   float fnumber = 90.0;
   char string[4] = "abc";
   vspf("%d %f %s", inumber, fnumber, string);
   printf("%s\n", buffer);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
vsprintf	+	+	+	+
vswprintf		+	+	+

1.1.2.15.95 vsscanf

Header File

io.h

Category

Memory and String Manipulation Routines

Prototype

```
int vsscanf(const char *buffer, const char *format, va_list arglist);
```

Description

Scans and formats input from a stream.

The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

Note: For details on format specifiers, see Scanf Format Specifiers.

vsscanf scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to vsscanf in the format string pointed to by format. Finally, vsscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields.

vsscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vsscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If vsscanf attempts to read at end-of-string, the return value is EOF.

Example

```
#include <stdio.h>
#include <stdarg.h>
char buffer[80] = "30 90.0 abc";
int vssf(char *fmt, ...)
{
   va_list  argptr;
   int cnt;
```

```
    fflush(stdin);
    va_start(argptr, fmt);
    cnt = vsscanf(buffer, fmt, argptr);
    va_end(argptr);
    return(cnt);
}
int main(void)
{
    int inumber;
    float fnumber;
    char string[80];
    vssf("%d %f %s", &inumber, &fnumber, string);
    printf("%d %f %s\n", inumber, fnumber, string);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

1.1.2.15.96 write

Header File

io.h

Category

Input/output Routines

Prototype

```
int write(int handle, void *buf, unsigned len);
```

Description

Writes to a file.

write writes a buffer of data to the file or device named by the given handle. handle is a file handle obtained from a creat, open, dup, or dup2 call.

This function attempts to write len bytes from the buffer pointed to by buf to the file associated with handle. Except when write is used to write to a text file, the number of bytes written to the file will be no more than the number requested. The maximum number of bytes that write can write is UINT_MAX -1, because UINT_MAX is the same as -1, which is the error return indicator for write. On text files, when write sees a linefeed (LF) character, it outputs a CR/LF pair. UINT_MAX is defined in limits.h.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disks or disk files, writing always proceeds from the current file pointer. For devices, bytes are sent directly to the device. For files opened with the O_APPEND option, the file pointer is positioned to EOF by write before writing the data.

Return Value

write returns the number of bytes written. A write to a text file does not count generated carriage returns. In case of error, write returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <string.h>
int main(void)
{
   int handle;
   char string[40];
   int length, res;
/*
Create a file named "TEST.$$$" in the current directory and write a string to it.  If
"TEST.$$$" already exists, it will be overwritten.
 */
   if ((handle = open("TEST.$$$", O_WRONLY | O_CREAT | O_TRUNC,
                         S_IREAD | S_IWRITE)) == -1)
   {
      printf("Error opening file.\n");
      exit(1);
   }
   strcpy(string, "Hello, world!\n");
   length = strlen(string);
   if ((res = write(handle, string, length)) != length)
   {
      printf("Error writing to the file.\n");
      exit(1);
   }
   printf("Wrote %d bytes to the file.\n", res);
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.15.97 SEEK_xxx #defines

Header File

io.h, stdio.h

Description

#defines that set seek starting points

Constant	Value	File Location
SEEK_SET	0	Seeks from beginning of file
SEEK_CUR	1	Seeks from current position
SEEK_END	2	Seeks from end of file

1.1.2.15.98 stderr, stdin, stdout

Header File

stdio.h

Description

Predefined streams automatically opened when the program is started.

Name	Meaning
stdin	Standard input device
stdout	Standard output device
stderr	Standard error output device

1.1.2.15.99 _F_xxxx #defines

Header File

stdio.h

Description

File status flags of streams

Name	Meaning
_F_RDWR	Read and write
_F_READ	Read-only file
_F_WRIT	Write-only file
_F_BUF	Malloc'ed buffer data
_F_LBUF	Line-buffered file
_F_ERR	Error indicator
_F_EOF	EOF indicator
_F_BIN	Binary file indicator
_F_IN	Data is incoming
_F_OUT	Data is outgoing
_F_TERM	File is a terminal

1.1.2.15.100 _IOxxx #defines

Header File

stdio.h

Description

Constants for defining buffering style to be used with a file.

Name	Meaning
_IOFBF	The file is fully buffered. When a buffer is empty, the next input operation will attempt to fill the entire buffer.
	On output, the buffer will be completely filled before any data is written to the file.

_IOLBF	The file is line buffered. When a buffer is empty, the next input operation will still attempt to fill the entire buffer.
	On output, however, the buffer will be flushed whenever a newline character is written to the file.
_IONBF	The file is unbuffered. The buf and size parameters of setbuf are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

1.1.2.15.101 BUFSIZ #define

Header File

stdio.h

Description

Default buffer size used by setbuf function.

1.1.2.15.102 EOF #define

Header File

stdio.h

Description

A constant indicating that end-of-file has been reached on a file.

1.1.2.15.103 L_ctermid #define

Header File

stdio.h

Description

The length of a device id string.

1.1.2.15.104 L_tmpnam #define

Header File

stdio.h

Description

Size of an array large enough to hold a temporary file name string.

1.1.2.15.105 TMP_MAX #define

Header File

stdio.h

Description

Maximum number of unique file names.

1.1.2.15.106 OPEN_MAX #define

Header File

stdio.h

Description

Number of files that can be open simultaneously.

Name	Meaning
FOPEN_MAX	Maximum files per process
SYS_OPEN	Maximum files for system

1.1.2.15.107 HANDLE_MAX #define

Header File

io.h

Description

Maximum number of handles.

1.1.2.16 limits.h

The following functions, macros, and classes are provided in `limits.h`:

1.1.2.16.1 CHAR_xxx #defines

Header File

limits.h

Description

Name	Meaning
CHAR_BIT	Type char, number of bits
CHAR_MAX	Type char, minimum value
CHAR_MIN	Type char, maximum value

These values are independent of whether type char is defined as signed or unsigned by default.

1.1.2.16.2 SCHAR_xxx #defines

Header File

limits.h

Description

Name	Meaning
SCHAR_MAX	Type char, maximum value
SCHAR_MIN	Type char, minimum value

1.1.2.16.3 Uxxxx_MAX #defines

Header File

limits.h

Description

Name	Maximum value for type xxx
UCHAR_MAX	unsigned char
USHRT_MAX	unsigned short
UINT_MAX	unsigned integer
ULONG_MAX	unsigned long

1.1.2.16.4 SHRT_xxx #defines

Header File

limits.h

Description

Name	Meaning
SHRT_MAX	Type short, maximum value
SHRT_MIN	Type short, minimum value

1.1.2.16.5 INT_xxx #defines

Header File

limits.h

Description

Maximum and minimum value for type int.

Name	Meaning
INT_MAX	Type int, maximum value
INT_MIN	Type int, minimum value

1.1.2.16.6 LONG_xxx #defines

Header File

limits.h

Description

Maximum and minimum value for type long.

Name	Meaning
LONG_MAX	Type long, maximum value
LONG_MIN	Type long, minimum value

1.1.2.17 locale.h

The following functions, macros, and classes are provided in `locale.h`:

1.1.2.17.1 localeconv

Header File

locale.h

Category

Miscellaneous Routines

Prototype

```
struct lconv *localeconv(void);
```

Description

Queries the locale for numeric format.

This function provides information about the monetary and other numeric formats for the current locale. The information is stored in a **struct** lconv type. The structure can only be modified by the setlocale. Subsequent calls to localeconv will update the lconv structure.

The lconv structure is defined in locale.h. It contains the following fields:

char	Decimal point used in nonmonetary formats. This can never be an empty string.
char	Separator used to group digits to the left of the decimal point. Not used with monetary quantities.
char	Size of each group of digits. Not used with monetary quantities. See the value listing table below.
char	International monetary symbol in the current locale. The symbol format is specified in the ISO 4217 Codes for the Representation of Currency and Funds.
char	Local monetary symbol for the current locale.
char	Decimal point used to format monetary quantities.
char	Separator used to group digits to the left of the decimal point for monetary quantities.
char	Size of each group of digits used in monetary quantities. See the value listing table below.
char	String indicating nonnegative monetary quantities.
char	String indicating negative monetary quantities.
char	Number of digits after the decimal point that are to be displayed in an internationally formatted monetary quantity.

char	Number of digits after the decimal point that are to be displayed in a formatted monetary quantity.
char	Set to 1 if currency_symbol precedes a nonnegative formatted monetary quantity. If currency_symbol is after the quantity, it is set to 0.
char	Set to 1 if currency_symbol is to be separated from the nonnegative formatted monetary quantity by a space. Set to 0 if there is no space separation.
char	Set to 1 if currency_symbol precedes a negative formatted monetary quantity. If currency_symbol is after the quantity, set to 0.
char	Set to 1 if currency_symbol is to be separated from the negative formatted monetary quantity by a space. Set to 0 if there is no space separation.
char	Indicate where to position the positive sign in a nonnegative formatted monetary quantity.
char	Indicate where to position the positive sign in a negative formatted monetary quantity.

Any of the above strings (except decimal_point) that is empty " " is not supported in the current locale. The nonstring **char** elements are nonnegative numbers. Any nonstring **char** element that is set to CHAR_MAX indicates that the element is not supported in the current locale.

The grouping and mon_grouping elements are set and interpreted as follows:

CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be used repeatedly for the remainder of the digits.
any other integer	Indicates how many digits make up the current group. The next element is read to determine the size of the next group of digits before the current group.

The p_sign_posn and n_sign_posn elements are set and interpreted as follows:

0	Use parentheses to surround the quantity and currency_symbol.
1	Sign string precedes the quantity and currency_symbol.
2	Sign string succeeds the quantity and currency_symbol.
3	Sign string immediately precedes the quantity and currency_symbol.
4	Sign string immediately succeeds the quantity and currency_symbol.

Return Value

Returns a pointer to the the filled-in structure of type **struct** lconv. The values in the structure will change whenever setlocale modifies the LC_MONETARY or LC_NUMERIC categories.

Example

```
#include <locale.h>
#include <stdio.h>
int main(void)
{
   struct lconv ll;
   struct lconv *conv = &ll;
/* read the locality conversion structure */
   conv = localeconv();
/* display the structure */
  printf("Decimal Point                : %s\n", conv-> decimal_point);
  printf("Thousands Separator          : %s\n", conv-> thousands_sep);
  printf("Grouping                     : %s\n", conv-> grouping);
  printf("International Currency symbol : %s\n", conv-> int_curr_symbol);
  printf("$ thousands separator        : %s\n", conv-> mon_thousands_sep);
  printf("$ grouping                   : %s\n", conv-> mon_grouping);
  printf("Positive sign                : %s\n", conv-> positive_sign);
```

```
  printf("Negative sign                : %s\n", conv-> negative_sign);
  printf("International fraction digits : %d\n", conv-> int_frac_digits);
  printf("Fraction digits              : %d\n", conv-> frac_digits);
  printf("Positive $ symbol precedes   : %d\n", conv-> p_cs_precedes);
  printf("Positive sign space separation: %d\n", conv-> p_sep_by_space);
  printf("Negative $ symbol precedes   : %d\n", conv-> n_cs_precedes);
  printf("Negative sign space separation: %d\n", conv-> n_sep_by_space);
  printf("Positive sign position       : %d\n", conv-> p_sign_posn);
  printf("Negative sign position       : %d\n", conv-> n_sign_posn);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.17.2 setlocale, _wsetlocale

Header File

locale.h

Category

Miscellaneous Routines

Prototype

```
char *setlocale(int category, const char *locale);

wchar_t * _wsetlocale( int category, const wchar_t *locale);
```

Description

Use the setlocale to select or query a locale.

C++ Builder supports all locales supported in Win95/98/2000 operating systems. See your system documentation for details.

The possible values for the category argument are as follows:

LC_ALL	Affects all the following categories
LC_COLLATE	Affects strcoll and strxfrm
LC_CTYPE	Affects single-byte character handling functions. The mbstowcs and mbtowc functions are not affected.
LC_MONETARY	Affects monetary formatting by the localeconv function
LC_NUMERIC	Affects the decimal point of non-monetary data formatting. This includes the printf family of functions, and the information returned by localeconv.
LC_TIME	Affects strftime

The locale argument is a pointer to the name of the locale or named locale category. Passing a NULL pointer returns the current locale in effect. Passing a pointer that points to a null string requests setlocale to look for environment variables to determine which locale to set. The locale names are **not** case sensitive.

When setlocale is unable to honor a locale request, the preexisting locale in effect is unchanged and a null pointer is returned.

If the locale argument is a NULL pointer, the locale string for the category is returned. If category is LC_ALL, a complete locale string is returned. The structure of the complete locale string consists of the names of all the categories in the current locale concatenated and separated by semicolons. This string can be used as the locale parameter when calling setlocale with any of

the LC_xxx values. This will reinstate all the locale categories that are named in the complete locale string, and allows saving and restoring of locale states. If the complete locale string is used with a single category, for example, LC_TIME, only that category will be restored from the locale string.

If an empty string "" is used as the locale parameter an implementation-defined locale is used. This is the ANSI C specified behavior.

To take advantage of dynamically loadable locales in your application, define _ _USELOCALES_ _ for each module. If _ _USELOCALES_ _ is not defined, all locale-sensitive functions and macros will work only with the default C locale.

If a NULL pointer is used as the argument for the locale parameter, setlocale returns a string that specifies the current locale in effect. If the category parameter specifies a single category, such as LC_COLLATE, the string pointed to will be the name of that category. If LC_ALL is used as the category parameter then the string pointed to will be a full locale string that will indicate the name of each category in effect.

```
.
.
.
localenameptr = setlocale( LC_COLLATE, NULL );
if (localenameptr)
printf( "%s\n", localenameptr );
.
.
.
```

The output here will be one of the module names together with the specified code page. For example, the output could be LC_COLLATE = English_United States.437.

```
.
.
.
localenameptr = setlocale( LC_ALL, NULL );
if (localenameptr)
printf( "%s\n", localenameptr );
.
.
.
```

An example of the output here could be the following:

```
LC_COLLATE=English_United States.437;
LC_TIME=English_United States.437;
LC_CTYPE=English_United States.437;
```

Each category in this full string is delimited by a semicolon. This string can be copied and saved by an application and then used again to restore the same locale categories at another time. Each delimited name corresponds to the locale category constants defined in locale.h. Therefore, the first name is the name of the LC_COLLATE category, the second is the LC_CTYPE category, and so on. Any other categories named in the locale.h header file are reserved for future implementation.

To set all default categories for the specified French locale:

```
setlocale( LC_ALL, "French_France.850" );
```

To find out which code page is currently being used:

```
localenameptr = setlocale( LC_ALL, NULL );
```

Return value

If selection is successful, setlocale returns a pointer to a string that is associated with the selected category (or possibly all categories) for the new locale.

If UNICODE is defined, _wsetlocale returns a **wchar_t** string.

On failure, a NULL pointer is returned and the locale is unchanged. All other possible returns are discussed in the Remarks section above.

Example

```
#include <locale.h>
#include <stdio.h>
int main(void)
{
   char *old_locale;
   /* The only locale supported in CodeGear C++ is "C" */
   old_locale = setlocale(LC_ALL,"C");
   printf("Old locale was %s\n",old_locale);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
setlocale	+	+	+	+
_wsetlocale		+		

1.1.2.18 malloc.h

The following functions, macros, and classes are provided in `malloc.h`:

1.1.2.18.1 alloca

Header File

malloc.h

Category

Memory Routines

Prototype

```
void *alloca(size_t size);
```

Description

Allocates temporary stack space.

alloca allocates size bytes on the stack; the allocated space is automatically freed up when the calling function exits.

The use of alloca is not encouraged. In the **try**-block of a C++ program the alloca function should never be used. If an exception

is thrown, any values placed on the stack by alloca will be corrupted.

Return Value

If enough stack space is available, alloca returns a pointer to the allocated stack area. Otherwise, it returns NULL.

Example

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
void test(int a)
{
   char *newstack;
   int len = a;
   char dummy[1];
dummy[0] = 0;          /* force good stack frame */
   printf("SP before calling alloca(0x%X) = 0x%X\n",len,_SP);
   newstack = (char *) alloca(len);
printf("SP after calling alloca = 0x%X\n",_SP);
   if (newstack)
      printf("Alloca(0x%X) returned %p\n",len,newstack);
   else
      printf("Alloca(0x%X) failed\n",len);
}
void main()
{
   test(256);
   test(16384);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.19 math.h

The following functions, macros, and classes are provided in `math.h`:

1.1.2.19.1 abs

Header File

stdlib.h, math.h

Category

Math Routines, Inline Routines

Prototype

```
int abs(int x);
```

Description

Returns the absolute value of an integer.

abs returns the absolute value of the integer argument x. If abs is called when stdlib.h has been included, it's treated as a macro that expands to inline code.

If you want to use the abs function instead of the macro, include

#undef abs

in your program, after the #include <stdlib.h>.

Return Value

The abs function returns an integer in the range of 0 to INT_MAX, with the exception that an argument with the value INT_MIN is returned as INT_MIN. The values for INT_MAX and INT_MIN are defined in header file limit.h.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
  int number = -1234;
  printf("number: %d  absolute value: %d\n", number, abs(number));
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.19.2 acos, acosl

Header File

math.h

Category

Math Routines

Prototype

```
double acos(double x);
```

```
long double acosl(long double x);
```

Description

Calculates the arc cosine.

acos returns the arc cosine of the input value.

acosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Arguments to acos and acosl must be in the range -1 to 1, or else acos and acosl return NAN and set the global variable errno to:

EDOM Domain error

Return Value

acos and acosl of an argument between -1 and +1 return a value in the range 0 to pi. Error handling for these routines can be modified through the functions _matherr_matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
```

```
int main(void)
{
  double result;
  double x = 0.5;
  result = acos(x);
  printf("The arc cosine of %lf is %lf\n", x, result);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
acos	+	+	+	+
acosl		+	+	+

1.1.2.19.3 asin, asinl

Header File

math.h

Category

Math Routines

Prototype

```
double asin(double x);
```

```
long double asinl(long double x);
```

Description

Calculates the arc sine.

asin of a real argument returns the arc sine of the input value.

asinl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Real arguments to asin and asinl must be in the range -1 to 1, or else asin and asinl return NAN and set the global variable errno to

EDOM Domain error

Return Value

asin and asinl of a real argument return a value in the range -pi/2 to pi/2. Error handling for these functions may be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = asin(x);
   printf("The arc sin of %lf is %lf\n", x, result);
  return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
asin	+	+	+	+
asinl		+	+	+

1.1.2.19.4 atan, atanl

Header File

math.h

Category

Math Routines

Prototype

```
double atan(double x);
```

```
long double atanl(long double x);
```

Description

Calculates the arc tangent.

atan calculates the arc tangent of the input value.

atanl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

atan and atanl of a real argument return a value in the range -pi/2 to pi/2. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = atan(x);
   printf("The arc tangent of %lf is %lf\n", x, result);
return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atan	+	+	+	+
atanl		+	+	+

1.1.2.19.5 atan2, atan2l

Header File

math.h

Category

Math Routines

Prototype

```
double atan2(double y, double x);
```

```
long double atan2l(long double y, long double x);
```

Description

Calculates the arc tangent of y/x.

atan2 returns the arc tangent of y/x; it produces correct results even when the resulting angle is near pi/2 or -pi/2 (x near 0). If both x and y are set to 0, the function sets the global variable errno to EDOM, indicating a domain error.

atan2l is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

atan2 and atan2l return a value in the range -pi to pi. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 90.0, y = 45.0;
   result = atan2(y, x);
   printf("The arc tangent ratio of %lf is %lf\n", (y / x), result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atan2	+	+	+	+
atan2l		+	+	+

1.1.2.19.6 atof, _wtof

Header File

stdlib.h, math.h

Category

Conversion Routines, Math Routines

Prototype

```
double atof(const char *s);
```

```
double _wtof(const wchar_t *s);
```

Description

Converts a string to a floating-point number.

- atof converts a string pointed to by s to **double**; this function recognizes the character representation of a floating-point number, made up of the following:

- An optional string of tabs and spaces
- An optional sign
- A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)
- An optional e or E followed by an optional signed integer

The characters must match this generic format:

[whitespace] [sign] [ddd] [.] [ddd] [e|E[sign]ddd]

atof also recognizes +INF and -INF for plus and minus infinity, and +NAN and -NAN for not-a-number.

In this function, the first unrecognized character ends the conversion.

The functions strtod and _strtold are similar to atof and provide better error detection, and hence are preferred in some applications.

Return Value

Returns the converted value of the input string.

If there is an overflow, atof returns plus or minus HUGE_VAL (or _LHUGE_VAL), errno is set to ERANGE (Result out of range), and _matherr (or _matherrl) is not called.

1.1.2.19.7 _atold, _wtold

Header File

math.h

Category

Conversion Routines, Math Routines

Prototype

```
long double _atold(const char *s);
long double _wtold(const wchar_t *s);
```

Description

Converts a string to a long double.

_wtold is the wide-character version. It converts a wide-character string to a long double.

_atof is the floating-point version of _atold.

_atold converts a string pointed to by s to a **long double**; this functions recognizes:

An optional string of tabs and spaces

An optional sign

A string of digits and an optional decimal point

An optional e or E followed by an optional signed integer

The first unrecognized character ends the conversion. There are no provisions for overflow.

The functions strtod and _strtold are similar to atof and _atold; they provide better error detection, and hence are preferred in some applications.

Return Value

Returns the converted value of the input string.

If there is an overflow, _atold returns plus or minus HUGE_VAL (or _LHUGE_VAL), errno is set to ERANGE (Result out of range), and _matherr (or _matherrl) is not called.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_atold		+		
_wtold		+		

1.1.2.19.8 cabs, cabsl

Header File

math.h

Category

Math Routines

Prototype

```
double cabs(struct complex z);
long double cabsl(struct _complexl z);
```

Description

cabs calculates the absolute value of a complex number. cabs is a macro that calculates the absolute value of z, a complex number. z is a structure with type complex; the structure is defined in math.h as

```
struct complex {
double x, y;
};
```

where x is the real part, and y is the imaginary part.

Calling cabs is equivalent to calling sqrt with the real and imaginary components of z, as shown here:

```
sqrt(z.x * z.x + z.y * z.y)
```

cabsl is the **long double** version; it takes a structure with type _complexl as an argument, and returns a **long double** result. The structure is defined in math.h as

```
struct _complexl {
long double x, y;
};
```

Return Value

cabs (or cabsl) returns the absolute value of z, a double. On overflow, cabs (or cabsl) returns HUGE_VAL (or _LHUGE_VAL) and sets the global variable errno to

ERANGE	Result out of range

Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
#ifdef __cplusplus
  #include <complex.h>
#endif
#ifdef __cplusplus /* if C++, use class complex */
  void print_abs(void)
  {
    complex<float> z(1.0, 2.0);
    double  absval;
    absval = abs(z);
    printf("The absolute value of %.2lfi %.2lfj is %.2lf",
      real(z), imag(z), absval);
  }
#else  /* below function is for C (and not C++) */
  void print_abs(void)
  {
    struct complex z;
    double absval;
    z.x = 2.0;
    z.y = 1.0;
    absval = cabs(z);
    printf("The absolute value of %.2lfi %.2lfj is %.2lf",
      z.x, z.y, absval);
  }
#endif
int main(void)
{
  print_abs();
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
cabs		+		
cabsl		+		

1.1.2.19.9 ceil, ceill

Header File

math.h

Category

Math Routines

Prototype

```
double ceil(double x);
```

```
long double ceill(long double x);
```

Description

Rounds up.

ceil finds the smallest integer not less than x.

ceill is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

These functions return the integer found as a **double** (ceil) or a **long double** (ceill).

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double number = 123.54;
   double down, up;
   down = floor(number);
   up = ceil(number);
   printf("original number     %5.2lf\n", number);
   printf("number rounded down %5.2lf\n", down);
   printf("number rounded up   %5.2lf\n", up);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ceil	+	+	+	+
ceill		+	+	+

1.1.2.19.10 cos, cosl

Header File

math.h

Category

Math Routines, Inline Routines

Prototype

```
double cos(double x);
```

```
long double cosl(long double x);
```

Description

Calculates the cosine of a value.

cos computes the cosine of the input value. The angle is specified in radians.

cosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

cos of a real argument returns a value in the range -1 to 1. Error handling for these functions can be modified through _matherr (or _matherrl).

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = cos(x);
   printf("The cosine of %lf is %lf\n", x, result);
   return 0;
```

```
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
cos	+	+	+	+
cosl		+	+	+

1.1.2.19.11 cosh, coshl

Header File

math.h

Category

Math Routines

Prototype

```
double cosh(double x);
```

```
long double coshl(long double x);
```

Description

Calculates the hyperbolic cosine of a value.

cosh computes the hyperbolic cosine:

coshl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

cosh returns the hyperbolic cosine of the argument.

When the correct value would create an overflow, these functions return the value HUGE_VAL (cosh) or _LHUGE_VAL (coshl) with the appropriate sign, and the global variable errno is set to ERANGE. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = cosh(x);
   printf("The hyperbolic cosine of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
cosh	+	+	+	+
coshl		+	+	+

1.1.2.19.12 exp, expl

Header File

math.h

Category

Math Routines

Prototype

```
double exp(double x);
```

```
long double expl(long double x);
```

Description

Calculates the exponential e to the x.

expl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

exp returns e to the x.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value overflows, exp returns the value HUGE_VAL and expl returns _LHUGE_VAL. Results of excessively large magnitude cause the global variable errno to be set to

ERANGE	Result out of range

On underflow, these functions return 0.0, and the global variable errno is not changed. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 4.0;
   result = exp(x);
   printf("'e' raised to the power \
          of %lf (e ^ %lf) = %lf\n",
          x, x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
exp	+	+	+	+
expl		+	+	+

1.1.2.19.13 fabs, fabsl

Header File

math.h

Category

Math Routines

Prototype

```
double fabs(double x);
```

```
long double fabsl(long double x);
```

Description

Returns the absolute value of a floating-point number.

fabs calculates the absolute value of x, a double. fabsl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

fabs and fabsl return the absolute value of x.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   float  number = -1234.0;

   printf("number: %f  absolute value: %f\n", number, fabs(number));
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fabs	+	+	+	+
fabsl		+	+	+

1.1.2.19.14 floor, floorl

Header File

math.h

Category

Math Routines

Prototype

```
double floor(double x);
```

```
long double floorl(long double x);
```

Description

Rounds down.

floor finds the largest integer not greater than x.

floorl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

floor returns the integer found as a **double**. floorl returns the integer found as a **long double**.

Portability

	POSIX	Win32	ANSI C	ANSI C++
floor	+	+	+	+
floorl		+	+	+

1.1.2.19.15 fmod, fmodl

Header File

math.h

Category

Math Routines

Prototype

```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
```

Description

Calculates x modulo y, the remainder of x/y.

fmod calculates x modulo y (the remainder f, where x = ay + f for some integer a, and 0 < f < y).

fmodl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

fmod and fmodl return the remainder f where x = ay + f (as described above). When y = 0, fmod and fmodl return 0.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double x = 5.0, y = 2.0;
   double result;
   result = fmod(x,y);
   printf("The remainder of (%lf / %lf) is %lf\n", x, y, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fmod	+	+	+	+
fmodl		+	+	+

1.1.2.19.16 frexp, frexpl

Header File

math.h

Category

Math Routines

Prototype

```
double frexp(double x, int *exponent);
```

```
long double frexpl(long double x, int *exponent);
```

Description

Splits a number into mantissa and exponent.

frexp calculates the mantissa m (a **double** greater than or equal to 0.5 and less than 1) and the integer value n such that x (the original **double** value) equals m * 2n. frexp stores n in the integer that exponent points to.

frexpl is the **long double** version; it takes a **long double** argument for x and returns a **long double** result.

Return Value

frexp and frexpl return the mantissa m. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double mantissa, number;
   int exponent;
   number = 8.0;
   mantissa = frexp(number, &exponent);
   printf("The number %lf is ", number);
   printf("%lf times two to the ", mantissa);
   printf("power of %d\n", exponent);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
frexp	+	+	+	+
frexpl		+	+	+

1.1.2.19.17 hypot, hypotl

Header File

math.h

Category

Math Routines

Prototype

```
double hypot(double x, double y);
```

```
long double hypotl(long double x, long double y);
```

Description

Calculates hypotenuse of a right triangle.

hypot calculates the value z where

z2 = x2 + y2 and z >= 0

This is equivalent to the length of the hypotenuse of a right triangle, if the lengths of the two sides are x and y.

hypotl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, these functions return z, a **double** (hypot) or a **long double**) (hypotl). On error (such as an overflow), they set the global variable errno to

ERANGE	Result out of range

and return the value HUGE_VAL (hypot) or _LHUGE_VAL) (hypotl). Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
 double result;
 double x = 3.0;
    double y = 4.0;
    result = hypot(x, y);
    printf("The hypotenuse is: %lf\n", result);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
hypot		+	+	+
hypotl		+	+	+

1.1.2.19.18 _i64toa, _i64tow

Header File

math.h, stdlib.h

Category

Conversion Routines, Math Routines,

Prototype

```
char *_i64toa(__int64 value, char *string, int radix);
```

```
wchar_t *_i64tow(__int64 value, wchar_t *string, int radix);
```

Description

_i64toa converts an **__int64** to a string. _i64tow is the unicode version. It converts a **__int64** to a wide-character string.

These functions convert value to a null-terminated string and store the result in string. value is an **__int64**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (**-**).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). These functions can return up to 33 bytes.

Return Value

Returns a pointer to string.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_i64toa		+		
_i64tow		+		

1.1.2.19.19 ldexp, ldexpl

Header File

math.h

Category

Math Routines

Prototype

```
double ldexp(double x, int exp);
```

```
long double ldexpl(long double x, int exp);
```

Description

Calculates x * 2^exp.

lexpl is the **long double** version; it takes a **long double** argument for x and returns a **long double** result.

Return Value

On success, ldexp (or ldexpl) returns the value it calculated, x * 2^exp. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double value;
    double x = 2;
    /* ldexp raises 2 by a power of 3
       then multiplies the result by 2   */
    value = ldexp(x,3);
    printf("The ldexp value is: %lf\n", value);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ldexp	+	+	+	+
ldexpl		+	+	+

1.1.2.19.20 ldiv

Header File

math.h

Category

Math Routines

Prototype

```
ldiv_t ldiv(long int numer, long int denom);
```

Description

Divides two **longs**, returning quotient and remainder.

ldiv divides two **longs** and returns both the quotient and the remainder as an ldiv_t type. numer and denom are the numerator and denominator, respectively.

The ldiv_t type is a structure of **long**s defined in stdlib.h as follows:

```
typedef struct {
long int quot; /* quotient */
long int rem; /* remainder */
} ldiv_t;
```

Return Value

ldiv returns a structure whose elements are quot (the quotient) and rem (the remainder).

Example

```
/* ldiv example */
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   ldiv_t lx;
   lx = ldiv(100000L, 30000L);
  printf("100000 div 30000 = %ld remainder %ld\n", lx.quot, lx.rem);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.19.21 log, logl

Header File

math.h

Category

Math Routines

Prototype

```
double log(double x);
long double logl(long double x);
```

Description

Calculates the natural logarithm of x.

log calculates the natural logarithm of x.

logl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

On success, log and logl return the value calculated, ln(x).

errno

EDOM	Domain error

If x is 0, the functions return the value negative HUGE_VAL (log) or negative _LHUGE_VAL (logl), and set errno to ERANGE. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double result;
   double x = 8.6872;
   result = log(x);
   printf("The natural log of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
log	+	+	+	+
logl		+	+	+

1.1.2.19.22 log10, log10l

Header File

math.h

Category

Math Routines

Prototype

```
double log10(double x);
```

```
long double log10l(long double x);
```

Description

log10 calculates the base ten logarithm of x.

log10l is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

On success, log10 (or log10l) returns the calculated value log base ten of x.

If the argument x passed to these functions is real and less than 0, the global variable errno is set to

EDOM	Domain error

If x is 0, these functions return the value negative HUGE_VAL (log10) or _LHUGE_VAL (log10l). Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double result;
   double x = 800.6872;
   result = log10(x);
   printf("The common log of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
log10	+	+	+	+
log10l		+	+	+

1.1.2.19.23 _matherr, _matherrl

Header File

math.h

Category

Diagnostic Routines, Math Routines

Prototype

```
int _matherr(struct _exception *e);
```

```
int _matherrl(struct _exceptionl *e);
```

Description

User-modifiable math error handler.

_matherr is called when an error is generated by the math library.

_matherrl is the **long double** version; it is called when an error is generated by the long double math functions.

_matherr and _matherrl each serve as a user hook (a function that can be customized by the user) that you can replace by

writing your own math error-handling routine.

_matherr and _matherrl are useful for information on trapping domain and range errors caused by the math functions. They do not trap floating-point exceptions, such as division by zero. See signal for information on trapping such errors.

You can define your own _matherr or _matherrl routine to be a custom error handler (such as one that catches and resolves certain types of errors); this customized function overrides the default version in the C library. The customized _matherr or _matherrl should return 0 if it fails to resolve the error, or nonzero if the error is resolved. When _matherr or _matherrl return nonzero, no error message is printed and the global variable errno is not changed.

Here are the _exception and _exceptionl structures (defined in math.h):

```
struct _exception {
int type;
char *name;
double arg1, arg2, retval;
};
struct _exceptionl {
int type;
char *name;
long double arg1, arg2, retval;
};
```

The members of the _exception and _exceptionl structures are shown in the following table:

type	The type of mathematical error that occurred; an enum type defined in the typedef _mexcep (see definition after this list).
name	A pointer to a null-terminated string holding the name of the math library function that resulted in an error.
arg1, arg2	The arguments (passed to the function that name points to) caused the error; if only one argument was passed to the function, it is stored in arg1.
retval	The default return value for _matherr (or _matherrl); you can modify this value.

The **typedef** _mexcep, also defined in math.h, enumerates the following symbolic constants representing possible mathematical errors:

DOMAIN	Argument was not in domain of function, such as log(-1).
SING	Argument would result in a singularity, such as pow(0, -2).
OVERFLOW	Argument would produce a function result greater than DBL_MAX (or LDBL_MAX), such as exp(1000).
UNDERFLOW	Argument would produce a function result less than DBL_MIN (or LDBL_MIN), such as exp(-1000).
TLOSS	Argument would produce function result with total loss of significant digits, such as sin(10e70).

The macros DBL_MAX, DBL_MIN, LDBL_MAX, and LDBL_MIN are defined in float.h

The source code to the default _matherr and _matherrl is on the C++Builder distribution disks.

The UNIX-style _matherr and _matherrl default behavior (printing a message and terminating) is not ANSI compatible. If you want a UNIX-style version of these routines, use MATHERR.C and MATHERRL.C provided on the C++Builder distribution disks.

Example

```
#include <math.h>
#include <string.h>
#include <stdio.h>
int _matherr (struct _exception *a)
{
  if (a->type == DOMAIN)
    if (!strcmp(a->name,"sqrt")) {
      a->retval = sqrt (-(a->arg1));
    return 1;
    }
  return 0;
}
int main(void)
{
  double x = -2.0, y;
  y = sqrt(x);
  printf("_Matherr corrected value: %lf\n",y);
  return 0;
}
```

1.1.2.19.24 modf, modfl

Header File

math.h

Category

Math Routines

Prototype

```
double modf(double x, double *ipart);
```

```
long double modfl(long double x, long double *ipart);
```

Description

Splits a **double** or **long double** into integer and fractional parts.

modf breaks the **double** x into two parts: the integer and the fraction. modf stores the integer in ipart and returns the fraction.

modfl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

modf and modfl return the fractional part of x.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double fraction, integer;
    double number = 100000.567;
    fraction = modf(number, &integer);
    printf("The whole and fractional parts of %lf are %lf and %lf\n",
           number, integer, fraction);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
modf	+	+	+	+

modfl		+	+	+

1.1.2.19.25 poly, polyl

Header File

math.h

Category

Math Routines

Prototype

```
double poly(double x, int degree, double coeffs[]);
```

```
long double polyl(long double x, int degree, long double coeffs[]);
```

Description

Generates a polynomial from arguments.

poly generates a polynomial in x, of degree degree, with coefficients coeffs[0], coeffs[1], ..., coeffs[degree]. For example, if n = 4, the generated polynomial is:

polyl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

poly and polyl return the value of the polynomial as evaluated for the given x.

Example

```
#include <stdio.h>
#include <math.h>
/* polynomial:  x**3 - 2x**2 + 5x - 1 */
int main(void)
{
  double array[] = { -1.0, 5.0, -2.0, 1.0
};
  double result;
  result = poly(2.0, 3, array);
  printf("The polynomial: x**3 - 2.0x**2 + 5x - 1 at 2.0 is %lf\n", result);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
poly		+		
polyl		+		

1.1.2.19.26 pow, powl

Header File

math.h

Category

Math Routines

Prototype

```
double pow(double x, double y);
long double powl(long double x, long double y);
```

Description

Calculates x to the power of y.

powl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, pow and powl return the value calculated of x to the power of y.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value would overflow, the functions return the value HUGE_VAL (pow) or _LHUGE_VAL (powl). Results of excessively large magnitude can cause the global variable errno to be set to

ERANGE	Result out of range

If the argument x passed to pow or powl is real and less than 0, and y is not a whole number, or if x is 0 and y is less than 0, or you call pow(0,0), the global variable errno is set to

EDOM	Domain error

Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  double x = 2.0, y = 3.0;
  printf("%lf raised to %lf is %lf\n", x, y, pow(x, y));
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
pow	+	+	+	+
powl		+	+	+

1.1.2.19.27 pow10, pow10l

Header File

math.h

Category

Math Routines

Prototype

```
double pow10(int p);
```

```
long double pow10l(int p);
```

Description

Calculates 10 to the power of p.

pow10l is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, pow10 returns the value calculated, 10 to the power of p and pow10l returns a **long double** result.

The result is actually calculated to **long double** accuracy. All arguments are valid, although some can cause an underflow or overflow.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  double p = 3.0;
  printf("Ten raised to %lf is %lf\n", p, pow10(p));
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
pow10		+		
pow10l		+		

1.1.2.19.28 sin, sinl

Header File

math.h

Category

Math Routines

Prototype

```
double sin(double x);
```

```
long double sinl(long double x);
```

Description

Calculates sine.

sin computes the sine of the input value. Angles are specified in radians.

sinl is the **long double** version; it takes a l**ong double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Return Value

sin and sinl return the sine of the input value.

Example

```
#include <stdio.h>
```

```
#include <math.h>
int main(void)
{
   double result, x = 0.5;
   result = sin(x);
   printf("The sin of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sin	+	+	+	+
sinl		+	+	+

1.1.2.19.29 sinh, sinhl

Header File

math.h

Category

Math Routines, Inline Routines

Prototype

```
double sinh(double x);
```

```
long double sinhl(long double x);
```

Description

Calculates hyperbolic sine.

sinh computes the hyperbolic sine.

sinl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for sinh and sinhl can be modified through the functions _matherr and _matherrl.

Return Value

sinh and sinhl return the hyperbolic sine of x.

When the correct value overflows, these functions return the value HUGE_VAL (sinh) or _LHUGE_VAL (sinhl) of appropriate sign. Also, the global variable errno is set to ERANGE.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x = 0.5;
   result = sinh(x);
   printf("The hyperbolic sin of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sinh	+	+	+	+
sinhl		+	+	+

1.1.2.19.30 sqrt, sqrtl

Header File

math.h

Category

Math Routines

Prototype

```
double sqrt(double x);
```

```
long double sqrtl(long double x);
```

Description

Calculates the positive square root.

sqrt calculates the positive square root of the argument x.

sqrtl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Return Value

On success, sqrt and sqrtl return the value calculated, the square root of x. If x is real and positive, the result is positive. If x is real and negative, the global variable errno is set to

EDOM	Domain error

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double x = 4.0, result;
   result = sqrt(x);
   printf("The square root of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sqrt	+	+	+	+
sqrtl		+	+	+

1.1.2.19.31 tan, tanl

Header File

math.h

Category

Math Routines

Prototype

```
double tan(double x);
```

```
long double tanl(long double x);
```

Description

Calculates the tangent.

tan calculates the tangent. Angles are specified in radians.

tanl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these routines can be modified through the functions _matherr and _matherrl..

Return Value

tan and tanl return the tangent of x, sin(x)/cos(x).

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x;
   x = 0.5;
   result = tan(x);
   printf("The tan of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tan	+	+	+	+
tanl		+	+	+

1.1.2.19.32 tanh, tanhl

Header File

math.h

Category

Math Routines

Prototype

```
double tanh(double x);
```

```
long double tanhl(long double x);
```

Description

Calculates the hyperbolic tangent.

tanh computes the hyperbolic tangent, sinh(x)/cosh(x).

tanhl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Return Value

tanh and tanhl return the hyperbolic tangent of x.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x;
   x = 0.5;
   result = tanh(x);
   printf("The hyperbolic tangent of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tanh	+	+	+	+
tanhl		+	+	+

1.1.2.19.33 HUGE_VAL #defines

Header File

math.h

Description

Overflow value for math functions.

1.1.2.19.34 M_E, M_LOGxxx, M_LNxx #defines

Header File

math.h

Description

The constant values for logarithm functions.

Name	Meaning
M_E	The value of e
M_LOG2E	The value of log(e)
M_LOG10E	The value of log10(e)
M_LN2	The value of ln(2)
M_LN10	The value of ln(10)

1.1.2.19.35 PI constants

Header File

math.h

Description

Common constants of pi

Name	Meaning
M_PI	pi
M_PI_2	One-half pi
M_PI_4	One-fourth pi
M_1_PI	One divided by pi
M_2_PI	Two divided by pi
M_1_SQRTPI	One divided by the square root of pi
M_2_SQRTPI	Two divided by the square root of pi

1.1.2.19.36 M_SQRTxx #defines

Header File

math.h

Description

Constant values for square roots of 2.

Name	Meaning
M_SQRT2	Square root of 2
M_SQRT_2	1/2 the square root of 2

1.1.2.20 mem.h

The following functions, macros, and classes are provided in `mem.h`:

1.1.2.20.1 memccpy

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines

Prototype

```
void *memccpy(void *dest, const void *src, int c, size_t n);
```

Description

Copies a block of n bytes.

memccpy is available on UNIX System V systems.

memccpy copies a block of n bytes from src to dest. The copying stops as soon as either of the following occurs:

- The character c is first copied into dest.
- n bytes have been copied into dest.

Return Value

memccpy returns a pointer to the byte in dest immediately following c, if c was copied; otherwise, memccpy returns NULL.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *src = "This is the source string";
   char dest[50];
   char *ptr;
   ptr = (char *) memccpy(dest, src, 'c', strlen(src));
   if (ptr)
   {
      *ptr = '\0';
      printf("The character was found: %s\n", dest);
   }
   else
      printf("The character wasn't found\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.20.2 memchr, _wmemchr

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines

Prototype

```
void *memchr(const void s, int c, size_t n);/ C only */
const void *memchr(const void *s, int c, size_t n);// C++ only
void *memchr(void *s, int c, size_t n);// C++ only
void *memchr(const void s, int c, size_t n);/ C and C++ */
void * _wmemchr(void s, int c, size_t n);/ unicode version */
```

Description

Searches n bytes for character c.

memchr is available on UNIX System V systems.

memchr searches the first n bytes of the block pointed to by s for character c.

Return Value

On success, memchr returns a pointer to the first occurrence of c in s; otherwise, it returns NULL.

Note: If you are using the intrinsic version of these functions, the case of n = 0 will return NULL.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char str[17];
    char *ptr;
    strcpy(str, "This is a string");
    ptr = (char *) memchr(str, 'r', strlen(str));
    if (ptr)
       printf("The character 'r' is at position: %d\n", ptr - str);
    else
       printf("The character was not found\n");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memchr	+	+	+	+
_wmemchr		+		

1.1.2.20.3 memcmp

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

Compares two blocks for a length of exactly n bytes.

memcmp is available on UNIX System V systems.

memcmp compares the first n bytes of the blocks s1 and s2 as **unsigned char**s.

Return Value

Because it compares bytes as **unsigned char**s, memcmp returns a value that is

- < 0 if s1 is less than s2
- = 0 if s1 is the same as s2

- > 0 if s1 is greater than s2

For example,

```
memcmp(“\xFF”, “\x7F”, 1)
```

returns a value greater than 0.

Note: If you are using the intrinsic version of these functions, the case of n = 0 will return NULL.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *buf1 = "aaa";
   char *buf2 = "bbb";
   char *buf3 = "ccc";
   int stat;
   stat = memcmp(buf2, buf1, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");
   stat = memcmp(buf2, buf3, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.20.4 memcpy, _wmemcpy

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
void *memcpy(void *dest, const void *src, size_t n);
```

```
void *_wmemcpy(void *dest, const void *src, size_t n);
```

Description

Copies a block of n bytes.

memcpy is available on UNIX System V systems.

memcpy copies a block of n bytes from src to dest. If src and dest overlap, the behavior of memcpy is undefined.

Return Value

memcpy returns dest.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char src[] = "******************************";
   char dest[] = "abcdefghijlkmnopqrstuvwxyz0123456709";
   char *ptr;
   printf("destination before memcpy: %s\n", dest);
   ptr = (char *) memcpy(dest, src, strlen(src));
   if (ptr)
      printf("destination after memcpy: %s\n", dest);
   else
      printf("memcpy failed\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memcpy	+	+	+	+
_wmemcpy		+		

1.1.2.20.5 memicmp

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines

Prototype

```
int memicmp(const void *s1, const void *s2, size_t n);
```

Description

Compares n bytes of two character arrays, ignoring case.

memicmp is available on UNIX System V systems.

memicmp compares the first n bytes of the blocks s1 and s2, ignoring character case (upper or lower).

Return Value

memicmp returns a value that is

- < 0 if s1 is less than s2
- = 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *buf1 = "ABCDE123";
```

```
    char *buf2 = "abcde456";
    int stat;
    stat = memicmp(buf1, buf2, 5);
    printf("The strings to position 5 are ");
    if (stat)
       printf("not ");
    printf("the same\n");
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.20.6 memmove

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines

Prototype

```
void *memmove(void *dest, const void *src, size_t n);
```

Description

Copies a block of n bytes.

memmove copies a block of n bytes from src to dest. Even when the source and destination blocks overlap, bytes in the overlapping locations are copied correctly.

Return Value

memmove returns dest.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char *dest = "abcdefghijklmnopqrstuvwxyz0123456789";
  char *src = "******************************";
  printf("destination prior to memmove: %s\n", dest);
  memmove(dest, src, 26);
  printf("destination after memmove:    %s\n", dest);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memmove	+	+	+	+
_fmemmove				

1.1.2.20.7 memset, _wmemset

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
void *memset(void *s, int c, size_t n);
```

```
void *_wmemset(void *s, int c, size_t n);
```

Description

Sets n bytes of a block of memory to byte c.

memset sets the first n bytes of the array s to the character c.

Return Value

memset returns s.

Example

```
#include <string.h>
#include <stdio.h>
#include <mem.h>
int main(void)
{
    char buffer[] = "Hello world\n";
    printf("Buffer before memset: %s\n", buffer);
    memset(buffer, '*', strlen(buffer) - 1);
    printf("Buffer after memset:  %s\n", buffer);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memset	+	+	+	+
_wmemset		+		

1.1.2.20.8 setmem

Header File

mem.h

Category

Memory and String Manipulation Routines

Prototype

```
void setmem(void *dest, unsigned length, char value);
```

Description

Assigns a value to a range of memory.

setmem sets a block of length bytes, pointed to by dest, to the byte value.

Return Value

None.

Example

```
#include <stdio.h>
#include <alloc.h>
#include <mem.h>
int main(void)
{
   char *dest;
   dest = (char *) calloc(21, sizeof(char));
   setmem(dest, 20, 'c');
   printf("%s\n", dest);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.21 new.h

The following functions, macros, and classes are provided in new.h:

1.1.2.21.1 set_new_handler function

Header File

new.h

Category

Memory Routines

Syntax

```
typedef void (new * new_handler)();
new_handler set_new_handler(new_handler my_handler);
```

Description

set_new_handler installs the function to be called when the global **operator new** or **operator new[]()** cannot allocate the requested memory. By default the new operators throw an bad_alloc exception if memory cannot be allocated. You can change this default behavior by calling set_new_handler to set a new handler. To retain the traditional version of new, which does not throw exceptions, you can use set_new_handler(0).

If new cannot allocate the requested memory, it calls the handler that was set by a previous call to set_new_handler. If there is no handler installed by set_new_handler, new returns 0. my_handler should specify the actions to be taken when new cannot satisfy a request for memory allocation. The new_handler type, defined in new.h, is a function that takes no arguments and returns **void**. A new_handler can throw a bad_alloc exception.

- The user-defined my_handler should do one of the following:

- return after freeing memory
- throw an bad_alloc exception or an exception derived from bad_alloc
- call abort or exit functions

If my_handler returns, then new will again attempt to satisfy the request.

Ideally, my_handler would free up memory and return. new would then be able to satisfy the request and the program would continue. However, if my_handler cannot provide memory for new, my_handler must throw an exception or terminate the program. Otherwise, an infinite loop will be created.

Preferably, you should overload **operator new()** and **operator new[]()** to take appropriate actions for your applications.

Return Value

set_new_handler returns the old handler, if one has been registered.

The user-defined argument function, my_handler, should not return a value.

Example

```
#include <iostream>
#include <new.h>
#include <stdlib.h>
using std::cout;
using std::hex;
void mem_warn() {
  std::cerr << "\nCan't allocate!";
  exit(1);
}

void main(void) {
  std::set_new_handler(mem_warn);
  char *ptr = new char[100];
  cout << "\nFirst allocation: ptr = " << hex << long(ptr);
  ptr = new char[64000U];
  cout << "\nFinal allocation: ptr = " << hex << long(ptr);
  std::set_new_handler(0);  // Reset to default.
}
```

1.1.2.21.2 _new_handler

Header File

new.h

Syntax

```
typedef void (*pvf)();

pvf _new_handler;
```

Description

_new_handler contains a pointer to a function that takes no arguments and returns **void**. If **operator new()** is unable to allocate the space required, it will call the function pointed to by _new_handler; if that function returns it will try the allocation again. By default, the function pointed to by _new_handler simply terminates the application. The application can replace this handler, however, with a function that can try to free up some space. This is done by assigning directly to _new_handler or by calling the function set_new_handler, which returns a pointer to the former handler.

As an alternative, you can set using the function set_new_handler, like this:

```
pvf set_new_handler(pvf p);
```

_new_handler is provided primarily for compatibility with C++ version 1.2. In most cases this functionality can be better provided

by overloading **operator new()**.

Example

1.1.2.22 process.h

The following functions, macros, and classes are provided in `process.h`:

1.1.2.22.1 _adopt_thread

Header File

process.h

Category

Process Control Routines

Prototype

```
_PTHREAD_ADOPTION_DATA _adopt_thread(void (_USERENTRY *__start_address)(void *), void *__arglist, int free_flag );
```

Description

“Adopts” a thread, created with the Windows API CreateThread function, to the C++Builder RTL by hooking up the necessary internal data (exceptions, stack info, and so forth). _adopt_thread thereby allows the RTL to handle exception issues in that thread. The execution path of this thread is then transferred to another function (the adoptive thread function). From the RTL's perspective, during the execution of this adoptive thread function, the thread appears as if it had been created by a call to _beginthreadex and is allowed all the benefits, such as calling other RTL functions and throwing/catching exceptions.

To create a thread, a user normally calls _beginthread. This hooks up the internal data automatically. _adopt_thread is primarily used in cases in which this internal data is not present. For example, this happens when a user is called from a thread that came from an outside source, such as ISAPI.

Using _adopt_thread thereby allows C++Builder compiled DLLs to be used from non-C++Builder EXEs. _adopt_thread works by:

- calling the user function (passing in the arglist param)
- unhooking the exception information
- returning a handle to the thread context

This process allows the same function to be used again (without reallocating all that data). At the end of this cycle, the _unadopt_thread function can be called to finally free up the rest of this data.

The last parameter, free_flag, determines whether the thread data structures are freed upon function exit:

- If free_flag is zero, then when the adoptive function exits only its exception handler is un-hooked. The rest of the RTL specific data that had been allocated during the thread's adoption, remains valid. If the same thread then calls _adopt_thread again, the existing data is used, the exception handler is rehooked, and the specified adoptive thread function is called.
- If free_flag is set to non-zero, the thread data structures will be freed before _adopt_thread returns. In this case the returned thread handle will be NULL since its associated date has already been freed.

Return Value

If the __free_flag parameter is **false** (zero), _adopt_thread returns a handle (thread context) that can later be used to free these data structures by passing it to _unadopt_thread().

If the __free_flag parameter to _adopt_thread is non-zero, the thread data is freed before _adopt_thread returns, and the

returned handle is NULL.

If an error has occurred, errno is set to:

ENOMEM Not enough memory

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.2 _beginthread

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned long _beginthread(void (_USERENTRY *__start)(void *), unsigned __stksize, void *__arg);
```

Description

Starts execution of a new thread.

Note: The start_address must be declared to be _USERENTRY.

The _beginthread function creates and starts a new thread. The thread starts execution at start_address.

The size of its stack in bytes is stack_size; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread is passed arglist as its only parameter; it can be NULL, but must be present. The thread function should terminate by simply returning; the _endthread. function will be called automatically. The _endthread function will automatically close the handle, and set the return value of the thread to zero.

Either this function or _beginthreadNT must be used instead of the operating system thread-creation API function because _beginthread and _beginthreadNT perform initialization required for correct operation of the runtime library functions.

This function is available only in the multithread libraries.

Return Value

_beginthread returns the handle of the new thread. The return value is a standard Windows handle that can be used in operating system API's such as SuspendThread and ResumeThread.

On error, the function returns -1, and the global variable errno is set to one of the following values:

EAGAIN	Too many threads
EINVAL	Invalid stack size (i.e. less than 16 bytes, or equal to zero)
ENOMEM	Not enough memory

Also see the description of the Win32 API GetLastError, in the MSDN Library.

Example

```
/*  Use the -tWM  (32-bit multi-threaded target) command-line switch for this example  */
```

```
#include <stdio.h>
#include <errno.h>
#include <stddef.h>     /* _threadid variable */
#include <process.h>    /* _beginthread, _endthread */
#include <time.h>       /* time, _ctime */
void thread_code(void *threadno)
{
    time_t t;
    time(&t);
    printf("Executing thread number %d, ID = %d, time = %s\n",
        (int)threadno, _threadid, ctime(&t));

}

void start_thread(int i)
{
    int thread_id;
#if     defined(__WIN32__)
    if ((thread_id = _beginthread(thread_code,4096,(void *)i)) == (unsigned long)-1)
#else
    if ((thread_id = _beginthread(thread_code,4096,(void *)i)) == -1)
#endif
    {
        printf("Unable to create thread %d, errno = %d\n",i,errno);
        return;
    }
    printf("Created thread %d, ID = %ld\n",i,thread_id);
}
int main(void)
{
    int i;
    for (i = 1; i < 20; i++)
        start_thread(i);
    printf("Hit ENTER to exit main thread.\n");
    getchar();
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.3 _beginthreadex

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned long _beginthreadex(void *__security_attr, unsigned __stksize, unsigned (__stdcall *__start)(void *), void *__arg, unsigned __create_flags, unsigned *__thread_id);
```

Description

Creates a thread and allows specifying the other parameters of the OS API CreateThread (such as security and thread creation flags). The _endthreadex function will be called automatically when the thread function terminates. The value returned from your thread function will be passed along to _endthreadex, which in turn will pass it along to the ExitThread API. The return value can

then be retrieved using the GetExitCodeThread API.

Unlike _endthread, the _endthreadex function does **not** close the thread handle, thereby allowing other threads to block on this one without fear that the handle will be freed out from under the system.

Other than the order of parameters and the closing of the thread handle, _beginthreadex performs same operation as _beginthreadNT.

Note: Note: The start address needs to be defined to return an unsigned, which is the thread exit code.

Return Value

_beginthreadex returns the handle of the new thread. The return value is a standard Windows handle that can be used in operating system API's such as SuspendThread and ResumeThread.

If unsuccessful, 0 is returned, and errno is set as follows:

EAGAIN	Too many threads
ENOMEM	Not enough memory
EINVAL	Bad stack value (i.e. less than 16 bytes or equal to zero)

Also see the description of the Win32 API GetLastError, in the MSDN Library.

Example

```
//*  Use the -tWM  (32-bit multi-threaded target) command-line switch for this example  */
#pragma checkoption -tWM
#include        <windows.h>
#include        <process.h>
#include        <stdio.h>
#define NTHREADS 25
/* This function acts as the 'main' function for each new thread */
static unsigned __stdcall threadMain(void *arg)
{
   printf("Thread %2d has an ID of %u\n", (int)arg, GetCurrentThreadId());
   return 0;
}
int main(void)
{
   HANDLE hThreads[NTHREADS];
   int i;
   unsigned threadId;
   SECURITY_ATTRIBUTES  sa = {
          sizeof(SECURITY_ATTRIBUTES), /* structure size */
          0,      /* No security descriptor */
          TRUE    /* Thread handle is inheritable */
   };
   /* Create NTHREADS inheritable threads that are initially suspended and that will run
starting at threadMain()*/
   for(i = 0; i < NTHREADS; i++) {
      hThreads[i] = (HANDLE)_beginthreadex(
         &sa,          /* Thread security */
         4096,         /* Thread stack size */
         threadMain,   /* Thread starting address */
         (void *)i,    /* Thread start argument */
         CREATE_SUSPENDED,  /* Create in suspended state */
         &threadId);   /* Thread ID */
      if(hThreads[i] == INVALID_HANDLE_VALUE) {
         MessageBox(0, "Thread Creation Failed", "Error", MB_OK);
         return  1;
      }
      printf("Created thread %2d with an ID of %u\n", i, threadId);
   }
```

```
   printf("\nPress ENTER to thaw all threads\n\n");
   getchar();

/* Resume the suspended threads */
   for(i = 0; i < NTHREADS; i++)
      ResumeThread(hThreads[i]);

/* Wait for the threads to run */
   WaitForMultipleObjects(NTHREADS, hThreads, TRUE, INFINITE);
/* Close all of the thread handles */
   for(i = 0; i < NTHREADS; i++)
      CloseHandle(hThreads[i]);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.4 _beginthreadNT

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned long _beginthreadNT(void (_USERENTRY *start_address)(void *), unsigned stack_size,
void *arglist, void *security_attrib, unsigned long create_flags, unsigned long *thread_id);
```

Description

Starts execution of a new thread under Windows NT.

Note: The start_address must be declared to be _USERENTRY.

All multithread Windows NT programs must use _beginthreadNT or the _beginthread function instead of the operating system thread-creation API function because these functions perform initialization required for correct operation of the runtime library functions. The _beginthreadNT function provides support for the operating system security. These functions are available only in the multithread libraries.

The _beginthreadNT function creates and starts a new thread. The thread starts execution at start_address. When your thread terminates, the _endthread function will be called automatically. _endthread will close the thread handle, and call the ExitThread API.

The size of its stack in bytes is stack_size; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread arglist can be NULL, but must be present.

The _beginthreadNT function uses the security_attr pointer to access the SECURITY_ATTRIBUTES structure. The structure contains the security attributes for the thread. If security_attr is NULL, the thread is created with default security attributes. The thread handle is not inherited if security_attr is NULL.

_beginthreadNT reads the create_flags variable for flags that provide additional information about the thread creation. This variable can be zero, specifying that the thread will run immediately upon creation. The variable can also be CREATE_SUSPENDED; in which case, the thread will not run until the ResumeThread function is called. ResumeThread is

provided by the Win32 API.

_beginthreadNT initializes the thread_id variable with the thread identifier.

Return Value

On success, _beginthreadNT returns the handle of the new thread. The return value is a standard Windows handle that can be used in operating system API's such as SuspendThread and ResumeThread.

On error, it returns -1, and the global variable errno is set to one of the following values:

EAGAIN	Too many threads
EINVAL	Invalid stack size (i.e. less than 16 bytes, or equal to zero)
ENOMEM	Not enough memory

Also see the description of the Win32 API GetLastError, in the MSDN Library.

Example

```
/*  Use the -tWM  (32-bit multi-threaded target) command-line switch for this example  */
#pragma checkoption -tWM
#include        <windows.h>
#include        <process.h>
#include        <stdio.h>
#define NTHREADS 25
static LONG runningThreads = 0;
static HANDLE doneEvent;
/* This function acts as the 'main' function for each new thread */
static void threadMain(void *arg)
{
  printf("Thread %2d has an ID of %u\n", (int)arg, GetCurrentThreadId());
  /* Use InterlockedDecrement() to modify the global runningThreads in a
 * thread safe manner.  When the count hits 0, signal the main thread if
 * it created an event for us.
 */
  if (InterlockedDecrement(&runningThreads) == 0 && doneEvent)
      SetEvent(doneEvent);
}
int main(void)
{
   HANDLE hThreads[NTHREADS];
   int i;
   DWORD  threadId;
   SECURITY_ATTRIBUTES  sa = {
          sizeof(SECURITY_ATTRIBUTES), /* structure size */
          0,      /* No security descriptor */
          TRUE    /* Thread handle is inheritable */
   };
   /* Create NTHREADS inheritable threads that are initially suspended and that will run
starting at threadMain()*/
   for(i = 0; i < NTHREADS; i++) {
      hThreads[i] = (HANDLE)_beginthreadNT(
         threadMain,   /* Thread starting address */
         4096,         /* Thread stack size */
         (void *)i,    /* Thread start argument */
         &sa,          /* Thread security */
         CREATE_SUSPENDED,  /* Create in suspended state */
         &threadId);   /* Thread ID */
      if(hThreads[i] == INVALID_HANDLE_VALUE) {
         MessageBox(0, "Thread Creation Failed", "Error", MB_OK);
         return  1;
      }
      ++runningThreads;
      printf("Created thread %2d with an ID of %u\n", i, threadId);
```

```
    }
    printf("\nPress ENTER to thaw all threads\n\n");
    getchar();
    /* Create the event that will signal when all threads are done */
    doneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

    /* Resume the suspended threads */
    for(i = 0; i < NTHREADS; i++)
       ResumeThread(hThreads[i]);
    /* Wait for all threads to finish execution, if we can */
    if (doneEvent) {
      WaitForSingleObject(doneEvent, INFINITE);
      CloseHandle(doneEvent);
    }
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.5 _c_exit

Header File

process.h

Category

Process Control Routines

Prototype

```
void _c_exit(void);
```

Description

Performs _exit cleanup without terminating the program.

_c_exit performs the same cleanup as _exit, except that it does not terminate the calling process.

Return Value

None.

Example

```
#include <process.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
main()
{
  int fd;
  char c;

  if ((fd = open("_c_exit.c",O_RDONLY)) < 0)
  {
    printf("Unable to open _c_exit.c for reading\n");
    return 1;
  }
  if (read(fd,&c,1) != 1)
    printf("Unable to read from open file handle %d before _c_exit\n",fd);
```

```
  else
    printf("Successfully read from open file handle %d before _c_exit\n",fd);
  _c_exit();
  if (read(fd,&c,1) != 1)
    printf("Unable to read from open file handle %d after _c_exit\n",fd);
  else
    printf("Successfully read from open file handle %d after _c_exit\n",fd);
 return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.6 _cexit

Header File

process.h

Category

Process Control Routines

Prototype

```
void _cexit(void);
```

Description

Performs exit cleanup without terminating the program.

_cexit performs the same cleanup as exit, closing all files but without terminating the calling process. The _cexit function calls any registered "exit functions" (posted with atexit). Before _cexit returns, it flushes all input/output buffers and closes all streams that were open.

Return Value

None.

Example

```
#include <windows.h>
#include <process.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
void exit_func(void)
{
  printf("Exit function called\n\n");
  printf("Close Window to return to program... It will beep if able to read from file");
}
int main(void)
{
  int fd;
  char c;
  if ((fd = open("_cexit.c",O_RDONLY)) < 0)
  {
printf("Unable to open _cexit.c for reading\n");
return 1;
  }
  atexit(exit_func);
```

```
  if (read(fd,&c,1) != 1)
printf("Unable to read from open file handle %d before _cexit\n",fd);
  else
printf("Successfully read from open file handle %d before _cexit\n",fd);
  _cexit();
  if (read(fd,&c,1) == 1)
  MessageBeep(0);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.7 cwait

Header File

process.h

Category

Process Control Routines

Prototype

```
int cwait(int *statloc, int pid, int action);
```

Description

Waits for child process to terminate.

The cwait function waits for a child process to terminate. The process ID of the child to wait for is pid. If statloc is not NULL, it points to the location where cwait will store the termination status. The action specifies whether to wait for the process alone, or for the process and all of its children.

If the child process terminated normally (by calling exit, or returning from main), the termination status word is defined as follows:

Bits 0-7	Zero
Bits 8-15	The least significant byte of the return code from the child process. This is the value that is passed to exit, or is returned from main. If the child process simply exited from main without returning a value, this value will be unpredictable.

If the child process terminated abnormally, the termination status word is defined as follows:

Bits 0-7Termination information about the child:

	1	Critical error abort.
	2	Execution fault, protection exception.
	3	External termination signal.
Bits 8-15	Zero	

Bits 8-15	Zero

If pid is 0, cwait waits for any child process to terminate. Otherwise, pid specifies the process ID of the process to wait for; this

value must have been obtained by an earlier call to an asynchronous spawn function.

The acceptable values for action are WAIT_CHILD, which waits for the specified child only, and WAIT_GRANDCHILD, which waits for the specified child and all of its children. These two values are defined in process.h.

Return Value

When cwait returns after a normal child process termination, it returns the process ID of the child.

When cwait returns after an abnormal child termination, it returns -1 to the parent and sets errno to EINTR (the child process terminated abnormally).

If cwait returns without a child process completion, it returns a -1 value and sets errno to one of the following values:

ECHILD	No child exists or the pid value is bad
EINVAL	A bad action value was specified

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.8 _endthread

Header File

process.h

Category

Process Control Routines

Prototype

```
void _endthread(void);
```

Description

Terminates execution of a thread.

The _endthread function terminates the currently executing thread by closing the thread handle and calling the ExitThread API. The thread must have been started by an earlier call to _beginthread or _beginthreadNT.. _endthread is called automatically by the runtime library when your thread function terminates.

This function is available in the multithread libraries; it is not in the single-threaded libraries.

Return Value

The function does not return a value.

1.1.2.22.9 _endthreadex

Header File

process.h

Category

Process Control Routines

Prototype

```
void _endthreadex(unsigned thread_retval);
```

Description

Terminates execution of the current thread by calling the ExitThread API, but without closing the handle. The thread must have been created by an earlier call to _beginthreadex. The runtime library will call _endthreadex autotmatically, when your thread function terminates. _endthreadex receives the return value of your thread function in thread_retval, and will pass it along to the Win32 ExitThread API.

Note: Note: Performs the same operation as _endthread(), but does not close the thread handle.

Return Value

None.

1.1.2.22.10 execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, _wexecl, _wexecle, _wexeclp, _wexeclpe, _wexecv, _wexecve, _wexecvp, _wexecvpe

Header File

process.h

Category

Process Control Routines

Prototype

```
int execl(char *path, char *arg0 *arg1, ..., *argn, NULL);
int _wexecl(wchar_t *path, wchar_t *arg0 *arg1, ..., *argn, NULL);
int execle(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);
int _wexecle(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t **env);
int execlp(char *path, char *arg0,*arg1, ..., *argn, NULL);
int _wexeclp(wchar_t *path, wchar_t *arg0,*arg1, ..., *argn, NULL);
int execlpe(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);
int _wexeclpe(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t **env);
int execv(char *path, char *argv[]);
int _wexecv(wchar_t *path, wchar_t *argv[]);
int execve(char *path, char *argv[], char **env);
int _wexecve(wchar_t *path, wchar_t *argv[], wchar_t **env);
int execvp(char *path, char *argv[]);
int _wexecvp(wchar_t *path, wchar_t *argv[]);
int execvpe(char *path, char *argv[], char **env);
int _wexecvpe(wchar_t *path, wchar_t *argv[], wchar_t **env);
```

Description

Loads and runs other programs.

The functions in the exec... family load and run (execute) other programs, known as child processes. When an exec... call succeeds, the child process overlays the parent process. There must be sufficient memory available for loading and executing the child process.

path is the file name of the called child process. The exec... functions search for path using the standard search algorithm:

- If no explicit extension is given, the functions search for the file as given. If the file is not found, they add .EXE and search again. If not found, they add .COM and search again. If found, the command processor, COMSPEC (Windows) or COMMAND.COM (DOS), is used to run the batch file.
- If an explicit extension or a period is given, the functions search for the file exactly as given.

The suffixes l, v, p, and e added to the exec... "family name" specify that the named function operates with certain capabilities.

l	specifies that the argument pointers (arg0, arg1, ..., argn) are passed as separate arguments. Typically, the l suffix is used when you know in advance the number of arguments to be passed.
v	specifies that the argument pointers (argv[0] ..., arg[n]) are passed as an array of pointers. Typically, the v suffix is used when a variable number of arguments is to be passed.
p	specifies that the function searches for the file in those directories specified by the PATH environment variable (without the p suffix, the function searches only the current working directory). If the path parameter does not contain an explicit directory, the function searches first the current directory, then the directories set with the PATH environment variable.
e	specifies that the argument env can be passed to the child process, letting you alter the environment for the child process. Without the e suffix, child processes inherit the environment of the parent process.

Each function in the exec... family must have one of the two argument-specifying suffixes (either l or v). The path search and environment inheritance suffixes (p and e) are optional; for example:

- execl is an exec... function that takes separate arguments, searches only the root or current directory for the child, and passes on the parent's environment to the child.
- execvpe is an exec... function that takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the env argument for altering the child's environment.

The exec... functions must pass at least one argument to the child process (arg0 or argv[0]); this argument is, by convention, a copy of path. (Using a different value for this 0th argument won't produce an error.)

path is available for the child process.

When the l suffix is used, arg0 usually points to path, and arg1, ..., argn point to character strings that form the new list of arguments. A mandatory null following argn marks the end of the list.

When the e suffix is used, you pass a list of new environment settings through the argument env. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

```
envvar = value
```

where envvar is the name of an environment variable, and value is the string value to which envvar is set. The last element in env is null. When env is null, the child inherits the parents' environment settings.

The combined length of arg0 + arg1 + ... + argn (or of argv[0] + argv[1] + ... + argn[n]), including space characters that separate the arguments, must be less than 260 bytes. Null terminators are not counted.

When an exec... function call is made, any open files remain open in the child process.

Return Value

If successful, the exec... functions do not return. On error, the exec... functions return -1, and the global variable errno is set to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Example

```
/* execl() example */
#include <stdio.h>
#include <process.h>

int main(int argc, char *argv[])
{
  int loop;
  printf("%s running...\n\n", argv[0]);

  if (argc == 1) {    /* check for only one command-line parameter */
    printf("%s calling itself again...\n", argv[0]);
    execl(argv[0], argv[0], "ONE", "TWO", "THREE", NULL);
    perror("EXEC:");
    exit(1);
  }
  printf("%s called with arguments:\n", argv[0]);
  for (loop = 1; loop <= argc; loop++)
    puts(argv[loop]);       /* Display all command-line parameters */
  return 0;
}
```

1.1.2.22.11 _expand

Header File

process.h

Category

Memory Routines

Prototype

```
void *_expand(void *block, size_t size);
```

Description

Grows or shrinks a heap block in place.

This function attempts to change the size of an allocated memory block without moving the block's location in the heap. The data in the block are not changed, up to the smaller of the old and new sizes of the block. The block must have been allocated earlier with malloc, calloc, or realloc, and must not have been freed.

Return Value

If _expand is able to resize the block without moving it, _expand returns a pointer to the block, whose address is unchanged. If _expand is unsuccessful, it returns a NULL pointer and does not modify or resize the block.

Example

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
```

```
void main(void)
{
  char *bufchar, *newbuf;
  printf( "Allocate a 512 element buffer\n" );
  if( (bufchar = (char *) calloc(512, sizeof( char ) )) == NULL)
    exit( 1 );
  printf( "Allocated %d bytes at %Fp\n",
    _msize ( bufchar ), (void __far *)bufchar );
  if ((newbuf = (char *) _expand (bufchar, 1024)) == NULL)
    printf ("cannot expand");
  else {
    bufchar = newbuf;
    printf (" Expanded block to %d bytes at %Fp\n",
      _msize( bufchar ) , (void __far *)bufchar );
  }
  free( bufchar );
  exit (0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.12 getpid

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned getpid(void)
```

Description

Gets the process ID of a program.

This function returns the current process ID--an integer that uniquely identifies the process.

Return Value

getpid returns the current process' ID.

Example

```
#include <stdio.h>
#include <process.h>
int main()
{
   printf("This program's process identification number (PID) "
          "number is %X\n", getpid());
   printf("Note: under DOS it is the PSP segment\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.22.13 spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, _wspawnl, _wspawnle, _wspawnlp, _wspawnlpe, _wspawnv, _wspawnve, _wspawnvp, _wspawnvpe

Header File

process.h

Category

Process Control Routines

Prototype

```
int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int _wspawnl(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);
int spawnle(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);
int _wspawnle(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL, wchar_t *envp[]);
int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int _wspawnlp(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);
int spawnlpe(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);
int _wspawnlpe(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL, wchar_t *envp[]);
int spawnv(int mode, char *path, char *argv[]);
int _wspawnv(int mode, wchar_t *path, wchar_t *argv[]);
int spawnve(int mode, char *path, char *argv[], char *envp[]);
int _wspawnve(int mode, wchar_t *path, wchar_t *argv[], wchar_t *envp[]);
int spawnvp(int mode, char *path, char *argv[]);
int _wspawnvp(int mode, wchar_t *path, wchar_t *argv[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
int _wspawnvpe(int mode, wchar_t *path, wchar_t *argv[], wchar_t *envp[]);
```

Note: In spawnle, spawnlpe, spawnv, spawnve, spawnvp, and spawnvpe, the last string must be NULL.

Description

The functions in the spawn... family create and run (execute) other files, known as child processes. There must be sufficient memory available for loading and executing a child process.

The value of mode determines what action the calling function (the parent process) takes after the spawn... call. The possible values of mode are

P_WAIT	Puts parent process on hold until child process completes execution.
P_NOWAIT	Continues to run parent process while child process runs. The child process ID is returned, so that the parent can wait for completion using cwait or wait.
P_NOWAITO	Identical to P_NOWAIT except that the child process ID isn't saved by the operating system, so the parent process can't wait for it using cwait or wait.
P_DETACH	Identical to P_NOWAITO, except that the child process is executed in the background with no access to the keyboard or the display.
P_OVERLAY	Overlays child process in memory location formerly occupied by parent. Same as an exec... call.

path is the file name of the called child process. The spawn... function calls search for path using the standard operating system search algorithm:

- If there is no extension or no period, they search for an exact file name. If the file is not found, they search for files first with the extension EXE, then COM, and finally BAT.
- If an extension is given, they search only for the exact file name.
- If only a period is given, they search only for the file name with no extension.
- If path does not contain an explicit directory, spawn... functions that have the **p** suffix search the current directory, then the directories set with the operating system PATH environment variable.

The suffixes p, l, and v, and e added to the spawn... "family name" specify that the named function operates with certain capabilities.

p	**The function searches for the file in those directories specified by the PATH environment variable. Without the p suffix, the function searches only the current working directory.**
l	The argument pointers arg0, arg1, ..., argn are passed as separate arguments. Typically, the l suffix is used when you know in advance the number of arguments to be passed.
v	The argument pointers argv[0], ..., arg[n] are passed as an array of pointers. Typically, the v suffix is used when a variable number of arguments is to be passed.
e	The argument envp can be passed to the child process, letting you alter the environment for the child process. Without the e suffix, child processes inherit the environment of the parent process.

Each function in the spawn... family must have one of the two argument-specifying suffixes (either l or v). The path search and environment inheritance suffixes (p and e) are optional.

For example:

- spawnl takes separate arguments, searches only the current directory for the child, and passes on the parent's environment to the child.
- spawnvpe takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the envp argument for altering the child's environment.

The spawn... functions must pass at least one argument to the child process (arg0 or argv[0]). This argument is, by convention, a copy of path. (Using a different value for this 0 argument won't produce an error.) If you want to pass an empty argument list to the child process, then arg0 or argv[0] must be NULL.

When the l suffix is used, arg0 usually points to path, and arg1, ..., argn point to character strings that form the new list of arguments. A mandatory null following argn marks the end of the list.

When the e suffix is used, you pass a list of new environment settings through the argument envp. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

```
envvar = value
```

where envvar is the name of an environment variable, and value is the string value to which envvar is set. The last element in

envp[] is null. When envp is null, the child inherits the parents' environment settings.

The combined length of arg0 + arg1 + ... + argn (or of argv[0] + argv[1] + ... + argv[n]), including space characters that separate the arguments, must be less than 260 bytes for Windows (128 for DOS). Null-terminators are not counted.

When a spawn... function call is made, any open files remain open in the child process.

Return Value

When successful, the spawn... functions, where mode is P_WAIT, return the child process' exit status (0 for a normal termination). If the child specifically calls exit with a nonzero argument, its exit status can be set to a nonzero value.

If mode is P_NOWAIT or P_NOWAITO, the spawn functions return the process ID of the child process. The ID obtained when using P_NOWAIT can be passed to cwait.

On error, the spawn... functions return -1, and the global variable errno is set to one of the following values:

E2BIG	Arg list too long
EINVAL	Invalid argument
ENOENT	Path or file name not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Example

```
#include <process.h>
#include <stdio.h>
void spawnl_example(void)
{
   int result;
   result = spawnl(P_WAIT, "bcc32.exe", "bcc32.exe", NULL);
   if (result == -1)
   {
      perror("Error from spawnl");
      exit(1);
   }
}
int main(void)
{
  spawnl_example();
  return 0;
}
```

1.1.2.22.14 _unadopt_thread

Header File

process.h

Category

Process Control Routines

Prototype

```
void _unadopt_thread(_PTHREAD_ADOPTION_DATA thd);
```

Description

Frees the RTL thread-specific data associated with a previous call to _adopt_thread.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.15 wait

Header File

process.h

Category

Process Control Routines

Prototype

```
int wait(int *statloc);
```

Description

Waits for one or more child processes to terminate.

The wait function waits for one or more child processes to terminate. The child processes must be those created by the calling program; wait cannot wait for grandchildren (processes spawned by child processes). If statloc is not NULL, it points to location where wait will store the termination status.

If the child process terminated normally (by calling exit, or returning from main), the termination status word is defined as follows:

Bits 0-7	Zero.
Bits 8-15	The least significant byte of the return code from the child process. This is the value that is passed to exit, or is returned from main. If the child process simply exited from main without returning a value, this value will be unpredictable. If the child process terminated abnormally, the termination status word is defined as follows:
Bits 0-7	Termination information about the child:

Bits 8-15	Zero.

Return Value

When wait returns after a normal child process termination it returns the process ID of the child.

When wait returns after an abnormal child termination it returns -1 to the parent and sets errno to EINTR.

If wait returns without a child process completion it returns a -1 value and sets errno to:

ECHILD	No child process exists

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.22.16 P_xxxx #defines

Header File

process.h

Description

Modes used by the spawn... functions.

Constant	Meaning
P_WAIT	Child runs separately, parent waits until exit
P_DETACH	Child and parent run concurrently with child process in background mode
P_NOWAIT	Child and parent run concurrently (Not implemented)
P_NOWAITO	Child and parent run concurrently, but the child process is not saved
P_OVERLAY	Child replaces parent so that parent no longer exists

1.1.2.23 setjmp.h

The following functions, macros, and classes are provided in `setjmp.h`:

1.1.2.23.1 longjmp

Header File

setjmp.h

Category

Miscellaneous Routines

Prototype

```
void longjmp(jmp_buf jmpb, int retval);
```

Description

Performs nonlocal goto.

A call to longjmp restores the task state captured by the last call to setjmp with the argument jmpb. It then returns in such a way that setjmp appears to have returned with the value retval.

A Win32 task state includes:

- Register variables
- EBX, EDI, ESI
- Stack pointer (ESP)
- Frame pointer (EBP)
- No segment registers are saved
- Flags are not saved

A task state is complete enough that setjmp and longjmp can be used to implement co-routines.

setjmp must be called before longjmp. The routine that called setjmp and set up jmpb must still be active and cannot have returned before the longjmp is called. If this happens, the results are unpredictable.

longjmp cannot pass the value 0; if 0 is passed in retval, longjmp will substitute 1.

Return Value

None.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
void subroutine(jmp_buf);
int main(void)
{

    int value;
    jmp_buf jumper;
    value = setjmp(jumper);
    if (value != 0)
    {
       printf("Longjmp with value %d\n", value);
       exit(value);
    }
    printf("About to call subroutine ... \n");
    subroutine(jumper);
    return 0;
}
void subroutine(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.23.2 setjmp

Header File

setjmp.h

Category

Miscellaneous Routines

Prototype

```
int setjmp(jmp_buf jmpb);
```

Description

Sets up for nonlocal goto.

setjmp captures the complete task state in jmpb and returns 0.

A later call to longjmp with jmpb restores the captured task state and returns in such a way that setjmp appears to have returned with the value val.

Under Win32, a task state includes:

- No segment registers are saved
- Register variables
- EBX, EDI, ESI
- Stack pointer (ESP)
- Frame pointer (EBP)
- Flags are not saved

A task state is complete enough that setjmp can be used to implement co-routines.

setjmp must be called before longjmp. The routine that calls setjmp and sets up jmpb must still be active and cannot have returned before the longjmp is called. If it has returned, the results are unpredictable.

setjmp is useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

Return Value

setjmp returns 0 when it is initially called. If the return is from a call to longjmp, setjmp returns a nonzero value (as in the example).

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
void subroutine(jmp_buf);
int main(void)
{

    int value;
    jmp_buf jumper;
    value = setjmp(jumper);
    if (value != 0)
    {
       printf("Longjmp with value %d\n", value);
       exit(value);
    }
    printf("About to call subroutine ... \n");
    subroutine(jumper);
    return 0;
}
void subroutine(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.24 share.h

The following functions, macros, and classes are provided in `share.h`:

1.1.2.24.1 SH_xxxx #defines

Header File

share.h

Description

File-sharing mode for use with sopen.

SH_COMPAT	Sets compatibility mode:
	Allows other opens with SH_COMPAT. The call will fail if the file has already been opened in any other shared mode.
SH_DENYNONE	Permits read/write access
	Allows other shared opens to the file, but not other SH_COMPAT opens
SH_DENYNO	Permits read/write access (provided for compatibility)
SH_DENYRD	Denies read access. Allows only writes from any other open to the file
SH_DENYRW	Denies read/write access. Only the current handle may have access to the file
SH_DENYWR	Denies write access. Allows only reads from any other open to the file
O_NOINHERIT	The file is not passed to child programs

These file-sharing attributes are in addition to any locking performed on the files.

1.1.2.25 signal.h

The following functions, macros, and classes are provided in `signal.h`:

1.1.2.25.1 raise

Header File

signal.h

Category

Process Control Routines

Prototype

```
int raise(int sig);
```

Description

Sends a software signal to the executing program.

raise sends a signal of type sig to the program. If the program has installed a signal handler for the signal type specified by sig, that handler will be executed. If no handler has been installed, the default action for that signal type will be taken.

The signal types currently defined in signal.h are noted here:

SIGABRT	Abnormal termination
SIGFPE	Bad floating-point operation
SIGILL	Illegal instruction
SIGINT	Ctrl-C interrupt
SIGSEGV	Invalid access to storage
SIGTERM	Request for program termination

SIGUSR1	User-defined signal
SIGUSR2	User-defined signal
SIGUSR3	User-defined signal
SIGBREAK	Ctrl-Break interrupt

Note: SIGABRT isn't generated by C++Builder during normal operation. It can, however, be generated by abort, raise, or unhandled exceptions.

Return Value

On success, raise returns 0.

On error it returns nonzero.

Example

```
#include <signal.h>
int main(void)
{
    int a, b;
    a = 10;
    b = 0;
    if (b == 0)
    /* preempt divide by zero error */
       raise(SIGFPE);
    a = a / b;
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.25.2 signal (C RTL)

Header File

signal.h

Category

Process Control Routines

Prototype

```
void (_USERENTRY *signal(int sig, void (_USERENTRY *func)(int sig[, int subcode])))(int);
```

Description

Specifies signal-handling actions.

signal determines how receipt of signal number sig will subsequently be treated. You can install a user-specified handler routine (specified by the argument func) or use one of the two predefined handlers, SIG_DFL and SIG_IGN, in signal.h. The function func must be declared with the _USERENTRY calling convention.

A routine that catches a signal (such as a floating point) also clears the signal. To continue to receive signals, a signal handler must be reinstalled by calling signal again.

SIG_DFL	Terminates the program
SIG_ERR	Indicates an error return from signal
SIG_IGN	Ignore this type signal

The following table shows signal types and their defaults:

SIGBREAK	Keyboard must be in raw mode.
SIGABRT	Abnormal termination. Default action is equivalent to calling _exit(3).
SIGFPE	Arithmetic error caused by division by 0, invalid operation, and the like. Default action is equivalent to calling _exit(1).
SIGILL	Illegal operation. Default action is equivalent to calling _exit(1).
SIGINT	Ctrl-C interrupt. Default action is equivalent to calling _exit(3).
SIGSEGV	Illegal storage access. Default action is equivalent to calling _exit(1).
SIGTERM	Request for program termination. Default action is equivalent to calling _exit(1).
	User-defined signals can be generated only by calling raise. Default action is to ignore the signal.

signal.h defines a type called sig_atomic_t, the largest integer type the processor can load or store atomically in the presence of asynchronous interrupts (this is a 32-bit integer -- a Borland C++ integer).

When a signal is generated by the raise function or by an external event, the following two things happen:

- If a user-specified handler has been installed for the signal, the action for that signal type is set to SIG_DFL.
- The user-specified function is called with the signal type as the parameter.

User-specified handler functions can terminate by a return or by a call to abort, _exit, exit, or longjmp. If your handler function is expected to continue to receive and handle more signals, you must have the handler function call signal again.

Borland C++ implements an extension to ANSI C when the signal type is SIGFPE, SIGSEGV, or SIGILL. The user-specified handler function is called with one or two extra parameters. If SIGFPE, SIGSEGV, or SIGILL has been raised as the result of an explicit call to the raise function, the user-specified handler is called with one extra parameter, an integer specifying that the handler is being explicitly invoked. The explicit activation values for SIGFPE, SIGSEGV and SIGILL are as follows

Note: Declarations of these types are defined in float.h.

SIGFPE	FPE_EXPLICITGEN
SIGSEGV	SEGV_EXPLICITGEN
SIGILL	ILL_EXPLICITGEN

If SIGFPE is raised because of a floating-point exception, the user handler is called with one extra parameter that specifies the FPE_xxx type of the signal. If SIGSEGV, SIGILL, or the integer-related variants of SIGFPE signals (FPE_INTOVFLOW or FPE_INTDIV0) are raised as the result of a processor exception, the user handler is called with two extra parameters:

1.The SIGFPE, SIGSEGV, or SIGILL exception type (see float.h for all these types). This first parameter is the usual ANSI signal type.

2.An integer pointer into the stack of the interrupt handler that called the user-specified handler. This pointer points to a list of the processor registers saved when the exception occurred. The registers are in the same order as the parameters to an interrupt function; that is, EBP, EDI, ESI, EDS, ES, EDX, ECX, EBX, EAX, EIP, CS, EFL. To have a register value changed when the

handler returns, change one of the locations in this list.

For example, to have a new SI value on return, do something like this:

```
*((int*)list_pointer + 2) = new_SI_value;
```

In this way, the handler can examine and make any adjustments to the registers that you want.

The following SIGFPE-type signals can occur (or be generated). They correspond to the exceptions that the 80x87 family is capable of detecting, as well as the "INTEGER DIVIDE BY ZERO" and the "INTERRUPT ON OVERFLOW" on the main CPU. (The declarations for these are in float.h.)

FPE_INTOVFLOW	INTO executed with OF flag set
FPE_INTDIV0	Integer divide by zero
FPE_INVALID	Invalid operation
FPE_ZERODIVIDE	Division by zero
FPE_OVERFLOW	Numeric overflow
FPE_UNDERFLOW	Numeric underflow
FPE_INEXACT	Precision
FPE_EXPLICITGEN	User program executed raise(SIGFPE)
FPE_STACKFAULT	Floating-point stack overflow or underflow
FPE_STACKFAULT	Stack overflow

The FPE_INTOVFLOW and FPE_INTDIV0 signals are generated by integer operations, and the others are generated by floating-point operations. Whether the floating-point exceptions are generated depends on the coprocessor control word, which can be modified with _control87. Denormal exceptions are handled by Borland C++ and not passed to a signal handler.

The following SIGSEGV-type signals can occur:

SEGV_BOUND	Bound constraint exception
SEGV_EXPLICITGEN	raise(SIGSEGV) was executed

The following SIGILL-type signals can occur:

ILL_EXECUTION	Illegal operation attempted
ILL_EXPLICITGEN	raise(SIGILL) was executed

When the signal type is SIGFPE, SIGSEGV, or SIGILL, a return from a signal handler is generally not advisable if the state of the floating point processor is corrupt, the results of an integer division are wrong, an operation that shouldn't have overflowed did, a bound instruction failed, or an illegal operation was attempted. The only time a return is reasonable is when the handler alters the registers so that a reasonable return context exists or the signal type indicates that the signal was generated explicitly (for example, FPE_EXPLICITGEN, SEGV_EXPLICITGEN, or ILL_EXPLICITGEN). Generally in this case you would print an error message and terminate the program using _exit, exit, or abort. If a return is executed under any other conditions, the program's action will probably be unpredictable.

Note: Take special care when using the signal function in a multithread program. The SIGINT, SIGTERM, and SIGBREAK signals can be used only by the main thread (thread one) in a non-Win32 application. When one of these signals occurs, the currently executing thread is suspended, and control transfers to the signal handler (if any) set up by thread one. Other signals can be handled by any thread.

Note: A signal handler should not use C++ runtime library functions, because a semaphore deadlock might occur. Instead, the handler should simply set a flag or post a semaphore, and return immediately.

Return Value

On success, signal returns a pointer to the previous handler routine for the specified signal type.

On error, signal returns SIG_ERR, and the external variable errno is set to EINVAL.

Example

```
/* signal example */
/*
    This example installs a signal handler routine for SIGFPE,
    catches an integer overflow condition, makes an adjustment to AX
    register, and returns.  This example program MAY cause your computer
    to crash, and will produce runtime errors depending on which memory
    model is used.
*/
#pragma inline
#include <stdio.h>
#include <signal.h>
#ifdef __cplusplus
   typedef void (*fptr)(int);
#else
   typedef void (*fptr)();
#endif
void Catcher(int *reglist)
{
   signal(SIGFPE, (fptr)Catcher);  //  ******reinstall signal handler
   printf("Caught it!\n"); *(reglist + 8) = 3; /* make return AX = 3 */
}
int main(void)
{
   signal(SIGFPE, (fptr)Catcher);  /* cast Catcher to appropriate type */
   asm     mov     ax,07FFFH       /* AX = 32767 */
   asm     inc     ax              /* cause overflow */
   asm     into                    /* activate handler */
   /* The handler set AX to 3 on return. If that had not happened,
      there would have been another exception when the next 'into'
      executed after the 'dec' instruction. */
   asm     dec     ax              /* no overflow now */
   asm     into                    /* doesn't activate */
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.25.3 SIG_xxx #defines

Header File

signal.h

Description

Predefined functions for handling signals generated by raise or by external events.

Name	Meaning
SIG_DFL	Terminate the program
SIG_IGN	No action, ignore signal

SIG_ERR	Return error code

1.1.2.25.4 SIGxxxx #defines

Header File

signal.h

Description

Signal types used by raise and signal.

Signal	Note	Meaning	Default Action
SIGABRT	(*)	Abnormal termination	= to calling _exit(3)
SIGFPE		Bad floating-point operation	= to calling _exit(1)
		Arithmetic error caused by	
		division by 0, invalid operation, etc.	
SIGILL		Illegal operation	= to calling _exit(1)
SIGINT		Control-C interrupt	= to calling _exit(3)
SIGSEGV		Invalid access to storage	= to calling _exit(1)
SIGTERM	(*)	Request for program termination	= to calling _exit(1)

(*) Signal types marked with a (*) aren't generated by Borland C++ during normal operation. However, they can be generated with raise.

1.1.2.26 stdarg.h

The following functions, macros, and classes are provided in `stdarg.h`:

1.1.2.26.1 va_arg, va_end, va_start

Header File

stdarg.h

Category

Variable Argument List Routines

Prototype

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Description

Implement a variable argument list.

Some C functions, such as vfprintf and vprintf, take variable argument lists in addition to taking a number of fixed (known) parameters. The va_arg, va_end, and va_start macros provide a portable way to access these argument lists. They are used for

stepping through a list of arguments when the called function does not know the number and types of the arguments being passed.

The header file stdarg.h declares one type (**va_list**) and three macros (va_start, va_arg, and va_end).

- **va_list**: This array holds information needed by va_arg and va_end. When a called function takes a variable argument list, it declares a variable ap of type **va_list**.
- **va_start**: This routine (implemented as a macro) sets ap to point to the first of the variable arguments being passed to the function. va_start must be used before the first call to va_arg or va_end.
- va_start takes two parameters: ap and lastfix. (ap is explained under va_list in the preceding paragraph; lastfix is the name of the last fixed parameter being passed to the called function.)
- **va_arg:** This routine (also implemented as a macro) expands to an expression that has the same type and value as the next argument being passed (one of the variable arguments). The variable ap to va_arg should be the same ap that va_start initialized.

Note: Because of default promotions, you cannot use **char, unsigned char**, or **float** types with va_arg.

Note: The first time va_arg is used, it returns the first argument in the list. Each successive time va_arg is used, it returns the next argument in the list. It does this by first dereferencing ap, and then incrementing ap to point to the following item. va_arg uses the type to both perform the dereference and to locate the following item. Each successive time va_arg is invoked, it modifies ap to point to the next argument in the list.

- **va_end:** This macro helps the called function perform a normal return. va_end might modify ap in such a way that it cannot be used unless va_start is recalled. va_end should be called after va_arg has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

Return Value

va_start and va_end return no values; va_arg returns the current argument in the list (the one that ap is pointing to).

Example

```
#include <stdio.h>
#include <stdarg.h>
/* calculate sum of a 0 terminated list */
void sum(char *msg, ...)
{
   int total = 0;
   va_list ap;
   int arg;
   va_start(ap, msg);
   while ((arg = va_arg(ap,int)) != 0) {
      total += arg;
   }
   printf(msg, total);
   va_end(ap);
}
int main(void) {
   sum("The total of 1+2+3+4 is %d\n", 1,2,3,4,0);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.27 stddef.h

The following functions, macros, and classes are provided in `stddef.h`:

1.1.2.27.1 offsetof

Header File

stddef.h

Category

Memory Routines

Prototype

```
size_t offsetof(struct_type, struct_member);
```

Description

Gets the byte offset to a structure member.

offsetof is available only as a macro. The argument struct_type is a **struct** type. struct_member is any element of the **struct** that can be accessed through the member selection operators or pointers.

If struct_member is a bit field, the result is undefined.

See also sizeof for more information on memory allocation and alignment of structures.

Return Value

offsetof returns the number of bytes from the start of the structure to the start of the named structure member.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

1.1.2.27.2 _threadid

Header File

stddef.h

Syntax

```
extern long _threadid;
```

Description

_threadid is a long integer that contains the ID of the currently executing thread. It is implemented as a macro, and should be declared only by including stddef.h.

1.1.2.27.3 NULL #define

Header File

stddef.h

Description

Null pointer constant that is compatible with any data object pointer. It is not compatible with function pointers. When a pointer is equivalent to NULL it is guaranteed not to point to any data object defined within the program.

1.1.2.28 stdio.h

The following functions, macros, and classes are provided in `stdio.h`:

1.1.2.29 stdlib.h

The following functions, macros, and classes are provided in `stdlib.h`:

1.1.2.29.1 abort

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
void abort(void);
```

Description

Abnormally terminates a program.

abort causes an abnormal program termination by calling raise(SIGABRT). If there is no signal handler for SIGABRT, then abort writes a termination message (Abnormal program termination) on stderr, then aborts the program by a call to _exit with exit code 3.

Return Value

abort returns the exit code 3 to the parent process or to the operating system command processor.

Example

```
#include <stdio.h>
#include <stdlib.h>
 int main(void)
 {
   printf("Calling abort()\n");
   abort();
   return 0; /* This is never reached */
 }
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.2 atexit

Header File

stdlib.h

Category

Process Control Routines

Prototype

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

Description

Registers termination function.

atexit registers the function pointed to by func as an exit function. Upon normal termination of the program, exit calls func just before returning to the operating system. fcmp must be used with the _USERENTRY calling convention.

Each call to atexit registers another exit function. Up to 32 functions can be registered. They are executed on a last-in, first-out basis (that is, the last function registered is the first to be executed).

Return Value

atexit returns 0 on success and nonzero on failure (no space left to register the function).

Example

```
#include <stdio.h>
#include <stdlib.h>
void exit_fn1(void)
{
   printf("Exit function #1 called\n");
}
void exit_fn2(void)
{
   printf("Exit function #2 called\n");
}
int main(void)
{
   /* post exit function #1 */
   atexit(exit_fn1);
   /* post exit function #2 */
   atexit(exit_fn2);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.3 atoi, _wtoi

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
int atoi(const char *s);
int _wtoi(const wchar_t *s);
```

Description

Converts a string to an integer.

- atoi converts a string pointed to by s to **int**; atoi recognizes (in the following order)
- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

[ws] [sn] [ddd]

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atoi (results are undefined).

Return Value

atoi returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**int**), atoi returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int n;
   char *str = "12345.67";
   n = atoi(str);
   printf("string = %s integer = %d\n", str, n);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atoi	+	+	+	+
_wtoi		+		

1.1.2.29.4 _atoi64, _wtoi64

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
__int64 _atoi64(const char *s);
```

```
__int64 _wtoi64(const wchar_t *s);
```

Description

Converts a string to an **__int64**.

The syntax of the string must be:

```
__int64 ::= [isspace]* [sign] digit [digit]*
```

Only decimal integers are acceptable.

_wtoi64 is the wide-character version. It converts a wide-character string to an **__int64**.

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atoi (results are undefined). There is no defined method to return an error indication to the caller. The result is undefined if the input string is invalid.

Return Value

Returns the converted value of the input string. If the string cannot be converted to a **__int64**, the return value is 0.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_atoi64		+		
_wtoi64		+		

1.1.2.29.5 atol, _wtol

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
long atol(const char *s);
long _wtol(const wchar_t *s);
```

Description

Converts a string to a long.

- atol converts the string pointed to by s to **long**. atol recognizes (in the following order)
- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

```
[ws] [sn] [ddd]
```

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atol (results are undefined).

Return Value

atol returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (b), atol returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   long l;
   char *lstr = "98765432";
   l = atol(lstr);
   printf("string = %s integer = %ld\n", lstr, l);
   return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atol	+	+	+	+
_wtol		+		

1.1.2.29.6 bsearch

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Binary search of an array.

bsearch searches a table (array) of nelem elements in memory, and returns the address of the first entry in the table that matches the search key. The array must be in order. If no match is found, bsearch returns 0.

Note: Because this is a binary search, the first matching entry is not necessarily the first entry in the table.

The type size_t is defined in stddef.h header file.

- nelem gives the number of elements in the table.
- width specifies the number of bytes in each table entry.

The comparison routine fcmp must be used with the _USERENTRY calling convention.

fcmp is called with two arguments: elem1 and elem2. Each argument points to an item to be compared. The comparison function compares each of the pointed-to items (*elem1 and *elem2), and returns an integer based on the results of the comparison.

- For bsearch, the fcmp return value is
- < 0 if *elem1 < *elem2
- == 0 if *elem1 == *elem2

- > 0 if *elem1 > *elem2

Return Value

bsearch returns the address of the first entry in the table that matches the search key. If no match is found, bsearch returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
typedef int (*fptr)(const void*, const void*);
#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))
int numarray[] = {123, 145, 512, 627, 800, 933};
int numeric (const int *p1, const int *p2)
{
   return(*p1 - *p2);
}
#pragma argsused
int lookup(int key)
{
   int  *itemptr;
   /* The cast of (int(*)(const void *,const void*))
      is needed to avoid a type mismatch error at
      compile time */
   itemptr = (int *) bsearch (&key, numarray, NELEMS(numarray),
      sizeof(int), (fptr)numeric);
   return (itemptr != NULL);
}
int main(void)
{
   if (lookup(512))
      printf("512 is in the table.\n");
   else
      printf("512 isn't in the table.\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.7 _crotl, _crotr

Header File

stdlib.h

Category

Math Routines

Prototype

```
unsigned char _crotl(unsigned char val, int count);
```

```
unsigned char _crotr(unsigned char val, int count);
```

Description

Rotates an unsigned char left or right.

_crotl rotates the given val to the left count bits. _crotr rotates the given val to the right count bits.

The argument val is an **unsigned char**, or its equivalent in decimal or hexadecimal form.

Return Value

- The functions return the rotated byte:
- _crotl returns the value of val left-rotated count bits.
- _crotr returns the value of val right-rotated count bits.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.8 div

Header File

stdlib.h

Category

Math Routines

Prototype

```
div_t div(int numer, int denom);
```

Description

Divides two integers, returning quotient and remainder.

div divides two integers and returns both the quotient and the remainder as a div_t type. numer and denom are the numerator and denominator, respectively. The div_t type is a structure of integers defined (with **typedef**) in stdlib.h as follows:

```
typedef struct {
int quot; /* quotient */
int rem; /* remainder */
} div_t;
```

Return Value

div returns a structure whose elements are quot (the quotient) and rem (the remainder).

Example

```
/* div example */
#include <stdlib.h>
#include <stdio.h>
div_t x;
int main(void)
{
   x = div(10,3);
   printf("10 div 3 = %d remainder %d\n",
   x.quot, x.rem);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.9 ecvt

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Description

Converts a floating-point number to a string.

ecvt converts value to a null-terminated string of ndig digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through dec (a negative value for dec means that the decimal lies to the left of the returned digits). There is no decimal point in the string itself. If the sign of value is negative, the word pointed to by sign is nonzero; otherwise, it's 0. The low-order digit is rounded.

Return Value

The return value of ecvt points to static data for the string of digits whose content is overwritten by each call to ecvt and fcvt.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char *string;
   double value;
   int dec, sign;
   int ndig = 10;
   value = 9.876;
   string = ecvt(value, ndig, &dec, &sign);
   printf("string = %s      dec = %d sign = %d\n", string, dec, sign);
   value = -123.45;
   ndig= 15;
   string = ecvt(value,ndig,&dec,&sign);
   printf("string = %s dec = %d sign = %d\n", string, dec, sign);
   value = 0.6789e5; /* scientific notation */
   ndig = 5;
   string = ecvt(value,ndig,&dec,&sign);
   printf("string = %s         dec = %d sign = %d\n", string, dec, sign);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.10 _exit

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
void _exit(int status);
```

Description

Terminates program.

_exit terminates execution without closing any files, flushing any output, or calling any exit functions.

The calling process uses status as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error.

Return Value

None.

Example

```
#include   <stdlib.h>
#include   <stdio.h>
void done(void);
int main(void)
{
   atexit(done);
   _exit(0);
   return 0;
}
void done()
{
   printf("hello\n");
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.29.11 exit

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
void exit(int status);
```

Description

Terminates program.

exit terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with atexit) are called.

status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit,

and a nonzero value indicates some error. It can be, but is not required, to be set with one of the following:

EXIT_FAILURE	Abnormal program termination; signal to operating system that program has terminated with an error
EXIT_SUCCESS	Normal program termination

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int status;
   printf("Enter either 1 or 2\n");
   status = getchar();
   exit(status - '0');
/* Note: this line is never reached */
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.12 fcvt

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Description

Converts a floating-point number to a string.

fcvt converts value to a null-terminated string digit starting with the leftmost significant digit with ndig digits to the right of the decimal point. fcvt then returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through dec (a negative value for dec means to the left of the returned digits). There is no decimal point in the string itself. If the sign of value is negative the word pointed to by sign is nonzero; otherwise it is 0.

The correct digit has been rounded for the number of digits to the right of the decimal point specified by ndig.

Return Value

The return value of fcvt points to static data whose content is overwritten by each call to fcvt and ecvt.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
```

```
{
  char *str;
  double num;
  int dec, sign, ndig = 5;
  /* a regular number */
  num = 9.876;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s decimal place = %d sign = %d\n", str, dec, sign);
  /* a negative number */
  num = -123.45;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s decimal place = %d sign = %d\n", str, dec, sign);
  /* scientific notation */
  num = 0.678e5;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s  decimal place= %d  sign = %d\n", str, dec, sign);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.13 _fullpath, _wfullpath

Header File

stdlib.h

Category

Directory Control Routines

Prototype

```
char * _fullpath(char *buffer, const char *path, int buflen);
```

```
wchar_t * _wfullpath(wchar_t *buffer, const wchar_t *path, int buflen);
```

Description

Converts a path name from relative to absolute.

_fullpath converts the relative path name in path to an absolute path name that is stored in the array of characters pointed to by buffer. The maximum number of characters that can be stored at buffer is buflen. The function returns NULL if the buffer isn't big enough to store the absolute path name or if the path contains an invalid drive letter.

If buffer is NULL, _fullpath allocates a buffer of up to _MAX_PATH characters. This buffer should be freed using free when it is no longer needed. _MAX_PATH is defined in stdlib.h.

Return Value

If successful the _fullpath function returns a pointer to the buffer containing the absolute path name.

On error, this function returns NULL.

Example

```
#include <stdio.h>
#include <stdlib.h>
char buf[_MAX_PATH];
void main(int argc, char *argv[])
{
```

```
  for ( ; argc; argv++, argc--)
  {
    if (_fullpath(buf, argv[0], _MAX_PATH) == NULL)
      printf("Unable to obtain full path of %s\n",argv[0]);
    else
      printf("Full path of %s is %s\n",argv[0],buf);
  }
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fullpath		+		
_wfullpath		NT only		

1.1.2.29.14 gcvt

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *gcvt(double value, int ndec, char *buf);
```

Description

Converts floating-point number to a string.

gcvt converts value to a null-terminated ASCII string and stores the string in buf. It produces ndec significant digits in FORTRAN F format, if possible; otherwise, it returns the value in the printf E format (ready for printing). It might suppress trailing zeros.

Return Value

gcvt returns the address of the string pointed to by buf.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char str[25];
   double num;
   int sig = 5; /* significant digits */
   /* a regular number */
   num = 9.876;
   gcvt(num, sig, str);
   printf("string = %s\n", str);
   /* a negative number */
   num = -123.4567;
   gcvt(num, sig, str);
   printf("string = %s\n", str);
   /* scientific notation */
   num = 0.678e5;
   gcvt(num, sig, str);
   printf("string = %s\n", str);
   return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.15 getenv, _wgetenv

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
char *getenv(const char *name);
```

```
wchar_t *_wgetenv(const wchar_t *name);
```

Description

Find or delete an environment variable from the system environment.

The environment consists of a series of entries that are of the form name=string\0.

getenv returns the value of a specified variable. name can be either uppercase or lowercase. name must not include the equal sign (**=**). If the specified environment variable does not exist, getenv returns a NULL pointer.

To delete the variable from the environment, use getenv("name=").

Note: Environment entries must not be changed directly. If you want to change an environment value, you must use putenv.

Return Value

On success, getenv returns the value associated with name.

If the specified name is not defined in the environment, getenv returns a NULL pointer.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *path, *ptr;
   int i = 0;
   /* get the current path environment */
   ptr = getenv("PATH");
   /* set up new path */
   path = (char *) malloc(strlen(ptr)+15);
   strcpy(path,"PATH=");
   strcat(path,ptr);
   strcat(path,";c:\\temp");
   /* replace the current path and display current environment */
   putenv(path);
   while (_environ[i])
       printf("%s\n",_environ[i++]);
   return 0;
```

```
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getenv	+	+	+	+
_wgetenv		+		

1.1.2.29.16 itoa, _itow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *itoa(int value, char *string, int radix);
```

```
wchar_t *_itow(int value, wchar_t *string, int radix);
```

Description

Converts an integer to a string.

itoa converts value to a null-terminated string and stores the result in string. With itoa, value is an integer. _itow is the unicode version of the function. It converts an integer to a wide-character string.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (-).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). itoa can return up to 33 bytes.

Return Value

itoa returns a pointer to string.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int number = 12345;
   char string[25];
   itoa(number, string, 10);
   printf("integer = %d string = %s\n", number, string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.17 labs

Header File

stdlib.h

Category

Math Routines

Prototype

```
long labs(long int x);
```

Description

Gives long absolute value.

labs computes the absolute value of the parameter x.

Return Value

labs returns the absolute value of x.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   long result;
   long x = -12345678L;
   result= labs(x);
   printf("number: %ld abs value: %ld\n", x, result);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.18 lfind

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void *lfind(const void *key, const void *base, size_t *num, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Performs a linear search.

lfind makes a linear search for the value of key in an array of sequential records. It uses a user-defined comparison routine fcmp.

The fcmp function must be used with the _USERENTRY calling convention.

The array is described as having *num records that are width bytes wide, and begins at the memory location pointed to by base.

Return Value

lfind returns the address of the first entry in the table that matches the search key. If no match is found, lfind returns NULL. The comparison routine must return 0 if *elem1 == *elem2, and nonzero otherwise (elem1 and elem2 are its two parameters).

Example

```
#include <stdio.h>
#include <stdlib.h>
int compare(int *x, int *y)
{
   return( *x - *y );
}
int main(void)
{
   int array[5] = {35, 87, 46, 99, 12};
   size_t nelem = 5;
   int key;
   int *result;
   key = 99;
   result = (int *) lfind(&key, array, &nelem,
      sizeof(int), (int(*)(const void *,const void *))compare);
   if (result)
      printf("Number %d found\n",key);
   else
      printf("Number %d not found\n",key);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.19 _lrotl, _lrotr

Header File

stdlib.h

Category

Math Routines

Prototype

```
unsigned long _lrotl(unsigned long val, int count);
```

```
unsigned long _lrotr(unsigned long val, int count);
```

Description

Rotates an **unsigned long** integer value to the left or right.

_lrotlrotates the given val to the left count bits. _lrotr rotates the given val to the right count bits.

Return Value

The functions return the rotated integer:

- _lrotl returns the value of val left-rotated count bits.
- _lrotr returns the value of val right-rotated count bits.

Example

```
#include <stdlib.h>
#include <stdio.h>
/* function prototypes */
int lrotl_example(void);
int lrotr_example(void);
/* lrotl example */
int lrotl_example(void)
{
   unsigned long result;
   unsigned long value = 100;
   result = _lrotl(value,1);
   printf("The value %lu rotated left one bit is: %lu\n", value, result);
   return 0;
}
/* lrotr example */
int lrotr_example(void)
{
   unsigned long result;
   unsigned long value = 100;
   result = _lrotr(value,1);
   printf("The value %lu rotated right one bit is: %lu\n", value, result);
   return 0;
}
int main(void)
{
   lrotl_example();
   lrotr_example();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.20 lsearch

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void *lsearch(const void *key, void *base, size_t *num, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Performs a linear search.

lsearch searches a table for information. Because this is a linear search, the table entries do not need to be sorted before a call to lsearch. If the item that key points to is not in the table, lsearch appends that item to the table.

- base points to the base (0th element) of the search table.

- num points to an integer containing the number of entries in the table.
- width contains the number of bytes in each entry.
- key points to the item to be searched for (the search key).

The function fcmp must be used with the _USERENTRY calling convention.

The argument fcmp points to a user-written comparison routine, that compares two items and returns a value based on the comparison.

To search the table, lsearch makes repeated calls to the routine whose address is passed in fcmp.

On each call to the comparison routine, lsearch passes two arguments:

key	a pointer to the item being searched for
elem	pointer to the element of base being compared.

fcmp is free to interpret the search key and the table entries in any way.

Return Value

lsearch returns the address of the first entry in the table that matches the search key.

If the search key is not identical to *elem, fcmp returns a nonzero integer. If the search key is identical to *elem, fcmp returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>     /* for strcmp declaration */
/* initialize number of colors */
char *colors[10] = { "Red", "Blue", "Green" };
int ncolors = 3;
int colorscmp(char **arg1, char **arg2)
{
   return(strcmp(*arg1, *arg2));
}
int addelem(char **key)
{
   int oldn = ncolors;
   lsearch(key, colors, (size_t *)&ncolors, sizeof(char *),
     (int(*)(const void *,const void *))colorscmp);
   return(ncolors == oldn);
}
int main(void)
{
   int i;
   char *key = "Purple";
   if (addelem(&key))
      printf("%s already in colors table\n", key);
   else
   {
      printf("%s added to colors table\n", key);
   }
   printf("The colors:\n");
   for (i = 0; i < ncolors; i++)
      printf("%s\n", colors[i]);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.21 _lrand

Header File

stdlib.h

Category

Math Routines

Prototype

```
long _lrand(void);
```

Description

_lrand is the long random number generator function. _rand uses a multiplicative congruential random number generator with period 2^64 to return successive pseudo-random numbers in the range from 0 to 2^31 - 1.

The generator is reinitialized by calling srand with an argument value of 1. It can be set to a new starting point by calling srand with a given seed number.

1.1.2.29.22 ltoa, _ltoa, _ltow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char * ltoa(long value, char * string, int radix);
char *_ltoa(long value, char *string, int radix);
wchar_t *_ltow(long value, wchar_t *string, int radix);
```

Description

Converts a **long** to a string. _ltow is the unicode version. It converts a **long** to a wide-charater string.

Converts value to a null-terminated string and stores the result in string. value is a **long**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (**-**).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). Can return up to 33 bytes.

Return Value

Returns a pointer to string.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char string[25];
```

```
    long value = 123456789L;
    ltoa(value,string,10);
    printf("number = %ld  string = %s\n", value, string);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ltoa		+		
_ltoa		+		
_ltow		+		

1.1.2.29.23 _makepath, _wmakepath

Header File

stdlib.h

Category

Directory Control Routines

Prototype

```
void _makepath(char *path, const char *drive, const char *dir, const char *name, const char *ext);
```

```
void _wmakepath(wchar_t *path, const wchar_t *drive, const wchar_t *dir, const wchar_t *name, const wchar_t *ext);
```

Description

Builds a path from component parts.

_makepath makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where

drive	=	X:
dir	=	\DIR\SUBDIR\
name	=	NAME
ext	=	.EXT
If drive is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.		
If dir is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.		
If name is empty or NULL, no file name is inserted in the path name.		
If ext is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.		
_makepath assumes there is enough space in path for the constructed path name. The maximum constructed length is _MAX_PATH. _MAX_PATH is defined in stdlib.h.		

_makepath and _splitpath are invertible; if you split a given path with _splitpath, then merge the resultant components with _makepath, you end up with path.		

If drive is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If dir is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If name is empty or NULL, no file name is inserted in the path name.

If ext is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

_makepath assumes there is enough space in path for the constructed path name. The maximum constructed length is _MAX_PATH. _MAX_PATH is defined in stdlib.h.

_makepath and _splitpath are invertible; if you split a given path with _splitpath, then merge the resultant components with _makepath, you end up with path.

Return Value

Example

```
#include <dir.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char s[_MAX_PATH];
  char drive[_MAX_DRIVE];
  char dir[_MAX_DIR];
  char file[_MAX_FNAME];
  char ext[_MAX_EXT];
  getcwd(s,_MAX_PATH);           /* get current working directory */
 if (s[strlen(s)-1] != '\\')
     strcat(s,"\\");             /* append a trailing \ character */
  _splitpath(s,drive,dir,file,ext); /* split the string to separate
    elems */
  strcpy(file,"DATA");
  strcpy(ext,".TXT");
  _makepath(s,drive,dir,file,ext); /* merge everything into one string */
  puts(s);                       /* display resulting string */
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_makepath		+		
_wmakepath		+		

1.1.2.29.24 max

Header File

stdlib.h

Category

C++ Prototyped Routines

Prototype

```
(type) max(a, b);
template <class T> T max( T t1, T t2 ); // C++ only
```

Description

Returns the larger of two values.

The C macro and the C++ template function compare two values and return the larger of the two. Both arguments and the routine declaration must be of the same type.

Return Value

max returns the larger of two values.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.25 mblen

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
int mblen(const char *s, size_t n);
```

Description

Determines the length of a multibyte character.

If s is not null, mblen determines the number of bytes in the multibyte character pointed to by s. The maximum number of bytes examined is specified by n.

The behavior of mblen is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If s is null, mblen returns a nonzero value if multibyte characters have state-dependent encodings. Otherwise, mblen returns 0.

If s is not null, mblen returns 0 if s points to the null character, and -1 if the next n bytes do not comprise a valid multibyte character; the number of bytes that comprise a valid multibyte character.

Example

```
#include <stdlib.h>
#include <stdio.h>
void main (void)
{
  int i ;
  char *mulbc = (char *)malloc( sizeof( char) );
  wchar_t widec = L'a';
  printf (" convert a wide character to multibyte character:\n" );
```

```
  i = wctomb (mulbc, widec);
  printf( "\tCharacters converted: %u\n", i);
  printf( "\tMultibyte character: %x\n\n", mulbc);
  printf( " Find length--in byte-- of multibyte character:\n");
  i = mblen( mulbc, MB_CUR_MAX);
  printf("\tLength--in bytes--if multiple character: %u\n",i);
  printf("\tWide character: %x\n\n", mulbc);
  printf( " Attempt to find length of a Wide character Null:\n");
  widec = L'\0';
  wctomb(mulbc, widec);
  i = mblen( mulbc, MB_CUR_MAX);
  printf("\tLength--in bytes--if multiple character: %u\n",i);
  printf("\tWide character: %x\n\n", mulbc);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.26 mbstowcs

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description

Converts a multibyte string to a **wchar_t** array.

The function converts the multibyte string s into the array pointed to by pwcs. No more than n values are stored in the array. If an invalid multibyte sequence is encountered, mbstowcs returns (size_t) -1.

The pwcs array will not be terminated with a zero value if mbstowcs returns n.

Return Value

If an invalid multibyte sequence is encountered, mbstowcs returns (size_t) -1. Otherwise, the function returns the number of array elements modified, not including the terminating code, if any.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int x;
  char    *mbst = (char *)malloc(MB_CUR_MAX);
  wchar_t *pwst = L"Hi";
  wchar_t *pwc      = (wchar_t *)malloc(sizeof( wchar_t));
  printf ("Convert to multibyte string:\n");
  x = wcstombs (mbst, pwst, MB_CUR_MAX);
  printf ("\tCharacters converted %u\n",x);
  printf ("\tHEx value of first");
  printf (" multibyte character: %#.4x\n\n", mbst);
  printf ("Convert back to wide character string:\n");
```

```
  x = mbstowcs(pwc, mbst, MB_CUR_MAX);
  printf( "\tCharacters converted: %u\n",x);
  printf( "\tHex value of first");
  printf( "wide character: %#.4x\n\n", pwc);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.27 mbtowc

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

Converts a multibyte character to **wchar_t** code.

If s is not null, mbtowc determines the number of bytes that comprise the multibyte character pointed to by s. Next, mbtowc determines the value of the type **wchar_t** that corresponds to that multibyte character. If there is a successful match between **wchar_t** and the multibyte character, and pwc is not null, the **wchar_t** value is stored in the array pointed to by pwc. At most n characters are examined.

Return Value

When s points to an invalid multibyte character, -1 is returned. When s points to the null character, 0 is returned. Otherwise, mbtowc returns the number of bytes that comprise the converted multibyte character.

The return value never exceeds MB_CUR_MAX or the value of n.

Example

```
#include <stdlib.h>
#include<stdio.h>
void main(void)
{
  int     x;
  char     *mbchar    = (char *)calloc(1, sizeof( char));
  wchar_t  wchar      = L'a';
  wchar_t   *pwcnull = NULL;
  wchar_t  *pwchar    = (wchar_t *)calloc(1,  sizeof( wchar_t ));
  printf ("Convert a wide character to multibyte character:\n");
  x = wctomb( mbchar, wchar);
  printf( "\tCharacters converted: %u\n", x);
  printf( "\tMultibyte character: %x\n\n", mbchar);
  printf ("Convert multibyte character back to a wide character:\n");
  x = mbtowc( pwchar, mbchar, MB_CUR_MAX );
  printf( "\tBytes converted: %u\n", x);
  printf( "\tWide character: %x\n\n", pwchar);
  printf ("Atempt to convert when target is NULL\n" );
  printf (" returns the length of the multibyte character:\n" );
  x = mbtowc (pwcnull, mbchar, MB_CUR_MAX );
```

```
printf ( "\tlength of multibyte character:%u\n\n", x );
printf ("Attempt to convert a NULL pointer to a" );
printf (" wide character:\n" );
mbchar = NULL;
x = mbtowc (pwchar, mbchar, MB_CUR_MAX);
printf( "\tBytes converted: %u\n", x );
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.28 min

Header File

stdlib.h

Category

C++ Prototyped Routines

Prototype

```
(type) min(a, b); /* macro version */
```

```
template <class T> T min( T t1, T t2 );// C++ only
```

Description

Returns the smaller of two values.

The C macro and the C++ template function compare two values and return the smaller of the two. Both arguments and the routine declaration must be of the same type.

Return Value

min returns the smaller of two values.

1.1.2.29.29 putenv, _wputenv

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
int putenv(const char *name);
```

```
int _wputenv(const wchar_t *name);
```

Description

Adds string to current environment.

putenv accepts the string name and adds it to the environment of the current process. For example,

```
putenv(“PATH=C:\\BC”);
```

putenv can also be used to modify an existing name. name can be either uppercase or lowercase. name must not include the equal sign (**=**). You can set a variable to an empty value by specifying an empty string on the right side of the '=' sign.

putenv can be used only to modify the current program's _environment. Once the program ends, the old _environment is restored. The _environment of the current process is passed to child processes, including any changes made by putenv.

Note that the string given to putenv must be static or global. Unpredictable results will occur if a local or dynamic string given to putenv is used after the string memory is released.

Return Value

On success, putenv returns 0; on failure, -1.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *path, *ptr;
   int i = 0;
   /* get the current path environment */
   ptr = getenv("PATH");
   /* set up new path */
   path = (char *) malloc(strlen(ptr)+15);
   strcpy(path,"PATH=");
   strcat(path,ptr);
   strcat(path,";c:\\temp");
   /* replace the current path and display current environment */
   putenv(path);
   while (_environ[i])
       printf("%s\n",_environ[i++]);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
putenv		+		
_wputenv		NT only		

1.1.2.29.30 qsort

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Sorts using the quicksort algorithm.

qsort is an implementation of the “median of three” variant of the quicksort algorithm. qsort sorts the entries in a table by repeatedly calling the user-defined comparison function pointed to by fcmp.

- base points to the base (0th element) of the table to be sorted.
- nelem is the number of entries in the table.
- width is the size of each entry in the table, in bytes.

fcmp, the comparison function, must be used with the _USERENTRY calling convention.

- fcmp accepts two arguments, elem1 and elem2, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (*elem1 and *elem2), and returns an integer based on the result of the comparison.
- *elem1 < *elem2 fcmp returns an integer < 0
- *elem1 == *elem2 fcmp returns 0
- *elem1 > *elem2 fcmp returns an integer > 0

In the comparison, the less-than symbol (<) means the left element should appear before the right element in the final, sorted sequence. Similarly, the greater-than (>) symbol means the left element should appear after the right element in the final, sorted sequence.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int sort_function( const void *a, const void *b);
char list[5][4] = { "cat", "car", "cab", "cap", "can" };
int main(void)
{
   int  x;
   qsort((void *)list, 5, sizeof(list[0]), sort_function);
   for (x = 0; x < 5; x++)
      printf("%s\n", list[x]);
   return 0;
}
int sort_function( const void *a, const void *b)
{
   return( strcmp((char *)a,(char *)b) );
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.31 rand

Header File

stdlib.h

Category

Math Routines

Prototype

```
int rand(void);
```

Description

Random number generator.

rand uses a multiplicative congruential random number generator with period 2 to the 32nd power to return successive pseudo-random numbers in the range from 0 to RAND_MAX. The symbolic constant RAND_MAX is defined in stdlib.h.

Return Value

rand returns the generated pseudo-random number.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    int i;
    randomize();
    printf("Ten random numbers from 0 to 99\n\n");
    for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.32 random

Header File

stdlib.h

Category

Math Routines

Prototype

```
int random(int num);
```

Description

Random number generator.

random returns a random number between 0 and (num-1). random(num) is a macro defined in stdlib.h. Both num and the random number returned are integers.

Return Value

random returns a number between 0 and (num-1).

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
/* prints a random number in the range 0 to 99 */
int main(void)
```

```
 {
    randomize();
    printf("Random number in the 0-99 range: %d\n", random (100));
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.33 randomize

Header File

stdlib.h, time.h

Category

Math Routines

Prototype

```
void randomize(void);
```

Description

Initializes random number generator.

randomize initializes the random number generator with a random value.

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
    int i;
    randomize();
    printf("Ten random numbers from 0 to 99\n\n");
    for(i=0; i<10; i++)
        printf("%d\n", rand() % 100);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.29.34 _rotl, _rotr

Header File

stdlib.h

Category

Math Routines

Prototype

```
unsigned short _rotl(unsigned short value, int count);
unsigned short _rotr(unsigned short value, int count);
```

Description

Bit-rotates an **unsigned** short integer value to the left or right.

_rotl rotates the given value to the left count bits.

_rotr rotates the given value to the right count bits.

Return Value

_rotl, and _rotr return the rotated integer:

- _rotl returns the value of value left-rotated count bits.
- _rotr returns the value of value right-rotated count bits.

Example

```
#include <stdlib.h>
#include <stdio.h>
/* rotl example */
int rotl_example(void)
{
   unsigned value, result;
   value = 32767;
   result = _rotl(value, 1);
   printf("The value %u rotated left one bit is: %u\n", value, result);
   return 0;
}
/* rotr example */
int rotr_example(void)
{
   unsigned value, result;
   value = 32767;
   result = _rotr(value, 1);
   printf("The value %u rotated right one bit is: %u\n", value, result);
   return 0;
}
int main(void)
{
   rotl_example();
   rotr_example();
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rotl		+		
_rotr				

1.1.2.29.35 _searchenv, _wsearchenv

Header File

stdlib.h

Category

Miscellaneous Routines

Prototype

```
char *_searchenv(const char *file, const char *varname, char *buf);
```

```
char *_wsearchenv(const wchar_t *file, const wchar_t *varname, wchar_t *buf);
```

Description

Looks for a file, using an environment variable as the search path.

_searchenv attempts to locate file, searching along the path specified by the operating system environment variable varname. Typical environment variables that contain paths are PATH, LIB, and INCLUDE.

_searchenv searches for the file in the current directory of the current drive first. If the file is not found there, the environment variable varname is fetched, and each directory in the path it specifies is searched in turn until the file is found, or the path is exhausted.

When the file is located, the full path name is stored in the buffer pointed to by buf. This string can be used in a call to access the file (for example, with fopen or exec...). The buffer is assumed to be large enough to store any possible file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at buf.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>

char buf[_MAX_PATH];

int main(void)
{
  /* ILINK32 will be found in your installation directory */
  _searchenv("ILINK32.EXE","PATH",buf);
  if (buf[0] == '\0')
    printf("ILINK32.EXE not found\n");
  else
    printf("ILINK32.EXE found in %s\n", buf);

  /* looks for nonexistent file */
  _searchenv("NOTEXIST.FIL","PATH",buf);
  if (buf[0] == '\0')
    printf("NOTEXIST.FIL not found\n");
  else
    printf("NOTEXIST.FIL found in %s\n", buf);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_searchenv		+		
_wsearchenv		NT only		

1.1.2.29.36 _searchstr, _wsearchstr

Header File

stdlib.h

Category

Miscellaneous Routines

Prototype

```
void _searchstr(const char *file, const char *ipath, char *buf);
```

```
void _wsearchstr(const wchar_t *file, const wchar_t *ipath,wchar_t *pathname);
```

Description

Searches a list of directories for a file.

_searchstr attempts to locate file, searching along the path specified by the string ipath.

_searchstr searches for the file in the current directory of the current drive first. If the file is not found there, each directory in ipath is searched in turn until the file is found, or the path is exhausted. The directories in ipath must be separated by semicolons.

When the file is located, the full path name is stored in the buffer pointed by by buf. This string can be used in a call to access the file (for example, with fopen or exec...). The buffer is assumed to be large enough to store any possible file name. The constant _MAX_PATH defined in stdlib.h, is the size of the largest file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at buf.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>
char buf[_MAX_PATH];
int main(void)
{
  /* look for ILINK32.EXE */
  _searchstr("ILINK32.EXE", "PATH", buf);
  if (buf[0] == '\0')
  printf ("ILINK32.EXE not found\n");
  else
  printf ("ILINK32.EXE found in %s\n", buf);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_searchstr		+		
_wsearchstr		NT only		

1.1.2.29.37 _splitpath, _wsplitpath

Header File

stdlib.h

Category

Directory Control Routines

Prototype

```
void _splitpath(const char *path, char *drive, char *dir, char *name, char *ext);
```

```
void _wsplitpath(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t *name, wchar_t *ext);
```

Description

Splits a full path name into its components.

_splitpath takes a file's full path name (path) as a string in the form

X:\DIR\SUBDIR\NAME.EXT

and splits path into its four components. It then stores those components in the strings pointed to by drive, dir, name, and ext. (All five components must be passed, but any of them can be a null, which means the corresponding component will be parsed but not stored.) The maximum sizes for these strings are given by the constants _MAX_DRIVE, _MAX_DIR, _MAX_PATH, _MAX_FNAME, and _MAX_EXT (defined in stdlib.h), and each size includes space for the null-terminator. These constants are defined in stdlib.h.

_MAX_PATH	path
_MAX_DRIVE	drive; includes colon (:)
_MAX_DIR	dir; includes leading and trailing backslashes (\)
_MAX_FNAME	name
_MAX_EXT	ext; includes leading dot (.)

_splitpath assumes that there is enough space to store each non-null component.

When _splitpath splits path, it treats the punctuation as follows:

- drive includes the colon (C:, A:, and so on).
- dir includes the leading and trailing backslashes (\BC\include\, \source\, and so on).
- name includes the file name.
- ext includes the dot preceding the extension (.C, .EXE, and so on).

_makepath and _splitpath are invertible; if you split a given path with _splitpath, then merge the resultant components with _makepath, you end up with path.

Return Value

None.

Example

```
#include <dir.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char s[_MAX_PATH];
  char drive[_MAX_DRIVE];
  char dir[_MAX_DIR];
  char file[_MAX_FNAME];
  char ext[_MAX_EXT];
  /* get current working directory */
```

```
  getcwd(s,_MAX_PATH);
  if (s[strlen(s)-1] != '\\')
  /* append a trailing \ character */
    strcat(s,"\\");
  /* split the string to separate elems */
  _splitpath(s,drive,dir,file,ext);
  strcpy(file,"DATA");
  strcpy(ext,".TXT");
  /* merge everything into one string */
  _makepath(s,drive,dir,file,ext);
  /* display resulting string */
  puts(s);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_splitpath		+		
_wsplitpath		+		

1.1.2.29.38 srand

Header File

stdlib.h

Category

Math Routines

Prototype

```
void srand(unsigned seed);
```

Description

Initializes random number generator.

The random number generator is reinitialized by calling srand with an argument value of 1. It can be set to a new starting point by calling srand with a given seed number.

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
   int i;
   time_t t;
   srand((unsigned) time(&t));
   printf("Ten random numbers from 0 to 99\n\n");
   for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.39 strlen, _mbslen, wcslen, _mbstrlen

Header File

string.h, mbstring.h, stdlib.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
size_t strlen(const char *s);
size_t wcslen(const wchar_t *s);
size_t _mbslen(const unsigned char *s);
size_t _mbstrlen(const char *s)
```

Description

Calculates the length of a string.

strlen calculates the length of s.

_mbslen and _mbstrlen test the string argument to determine the number of multibyte characters they contain.

_mbstrlen is affected by the LC_CTYPE category setting as determined by the setlocale function. The function tests to determine whether the string argument is a valid multibyte string.

_mbslen is affected by the code page that is in use. This function doesn't test for multibyte validity.

Return Value

strlen returns the number of characters in s, not counting the null-terminating character.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "CodeGear";
   printf("%d\n", strlen(string));
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strlen	+	+	+	+
_mbslen		+		
wcslen		+	+	+
_mbstrlen		+		

1.1.2.29.40 strtod, _strtold, wcstod, _wcstold

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
double strtod(const char *s, char **endptr);
double wcstod(const wchar_t *s, wchar_t **endptr);
long double _strtold(const char *s, char **endptr);
long double _wcstold(const wchar_t *s, wchar_t **endptr);
```

Description

Convert a string to a **double** or **long double** value.

strtod converts a character string, s, to a **double** value. s is a sequence of characters that can be interpreted as a **double** value; the characters must match this generic format:

```
[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
```

where:

[ws]	=	optional whitespace
[sn]	=	optional sign (+ or -)
[ddd]	=	optional digits
[fmt]	=	optional e or E
[.]	=	optional decimal point

strtod also recognizes +INF and -INF for plus and minus infinity, and +NAN and -NAN for not-a-number.

For example, here are some character strings that strtod can convert to **double**:

```
+ 1231.1981 e-1
502.85E2
+ 2010.952
```

strtod stops reading the string at the first character that cannot be interpreted as an appropriate part of a **double** value.

If endptr is not null, strtod sets *endptr to point to the character that stopped the scan (*endptr = &stopper). endptr is useful for error detection.

_strtold is the l**ong double** version; it converts a string to a **long double** value.

Return Value

These functions return the value of s as a **double** (strtod) or a **long double** (_strtold). In case of overflow, they return plus or minus HUGE_VAL (strtod) or _LHUGE_VAL (_strtold).

Example

```
#include <stdio.h>
```

```
#include <stdlib.h>
int main(void)
{
   char input[80], *endptr;
   double value;
   printf("Enter a floating point number:");
   gets(input);
   value = strtod(input, &endptr);
   printf("The string is %s the number is %lf\n", input, value);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtod	+	+	+	+
_strtold		+		
wcstod		+	+	+
_wcstold		+		

1.1.2.29.41 strtol, wcstol

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
long strtol(const char *s, char **endptr, int radix);
```

```
long wcstol(const wchar_t *s, wchar_t **endptr, int radix);
```

Description

Converts a string to a **long** value.

strtol converts a character string, s, to a **long** integer value. s is a sequence of characters that can be interpreted as a **long** value; the characters must match this generic format:

```
[ws] [sn] [0] [x] [ddd]
```

where:

[ws]	=	optional whitespace
[sn]	=	optional sign (+ or -)
[0]	=	optional zero (0)
[x]	=	optional x or X
[ddd]	=	optional digits

strtol stops reading the string at the first character it doesn't recognize.

If radix is between 2 and 36, the long integer is expressed in base radix. If radix is 0, the first few characters of s determine the base of the value being converted.

If radix is 1, it is considered to be an invalid value. If radix is less than 0 or greater than 36, it is considered to be an invalid value.

Any invalid value for radix causes the result to be 0 and sets the next character pointer *endptr to the starting string pointer.

If the value in s is meant to be interpreted as octal, any character other than 0 to 7 will be unrecognized.

If the value in s is meant to be interpreted as decimal, any character other than 0 to 9 will be unrecognized.

If the value in s is meant to be interpreted as a number in any other base, then only the numerals and letters used to represent numbers in that base will be recognized. (For example, if radix equals 5, only 0 to 4 will be recognized; if radix equals 20, only 0 to 9 and A to J will be recognized.)

If endptr is not null, strtol sets *endptr to point to the character that stopped the scan (*endptr = &stopper).

Return Value

strtol returns the value of the converted string, or 0 on error.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char *string = "87654321", *endptr;
   long lnumber;
   /* strtol converts string to long integer  */
   lnumber = strtol(string, &endptr, 10);
   printf("string = %s  long = %ld\n", string, lnumber);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtol	+	+	+	+
wcstol		+	+	+

1.1.2.29.42 strtoul, wcstoul

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
unsigned long strtoul(const char *s, char **endptr, int radix);
```

```
unsigned long wcstoul(const wchar_t *s, wchar_t **endptr, int radix);
```

Description

Converts a string to an **unsigned long** in the given radix.

strtoul operates the same as strtol, except that it converts a string str to an **unsigned long** value (where strtol converts to a **long**). Refer to the entry for strtol for more information.

Return Value

strtoul returns the converted value, an **unsigned long**, or 0 on error.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char *string = "87654321", *endptr;
   unsigned long lnumber;
   lnumber = strtoul(string, &endptr, 10);
   printf("string = %s  long = %lu\n",
   string, lnumber);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtoul	+	+	+	+
wcstoul		+	+	+

1.1.2.29.43 swab

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void swab(char *from, char *to, int nbytes);
```

Description

Swaps bytes.

swab copies nbytes bytes from the from string to the to string. Adjacent even- and odd-byte positions are swapped. This is useful for moving data from one machine to another machine with a different byte order. nbytes should be even.

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char source[15] = "rFna koBlrna d";
char target[15];
int main(void)
{
   swab(source, target, strlen(source));
   printf("This is target: %s\n", target);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

1.1.2.29.44 system, _wsystem

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
int system(const char *command);
int _wsystem(const wchar_t *command);
```

Description

Issues an operating system command.

system invokes the operating system command processor to execute an operating system command, batch file, or other program named by the string command, from inside an executing C program.

To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.

The COMSPEC environment variable is used to find the command processor program, so it need not be in the current directory.

Return Value

If command is a NULL pointer, system returns nonzero if a command processor is available.

If command is not a NULL pointer, system returns 0 if the command processor was successfully started.

If an error occurred, a -1 is returned and errno is set to one of the following:

ENOENT	Path or file function not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   printf("About to spawn a command-line program.\n");
   system("dir");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
system	+	+		
_wsystem		NT only		

1.1.2.29.45 _i64toa, _ui64tow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *_ui64toa(unsigned __int64 value, char *string, int radix);
wchar_t *_ui64tow(unsigned __int64 value, wchar_t *string, int radix);
```

Description

_ui64toa converts an unsigned __**int64** to a string.

_ui64tow is the unicode version. _ui64tow converts an unsigned __**int64** to a wide-character string.

These functions convert value to a null-terminated string and store the result in string. value is an __**int64**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (**-**).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). Can return up to 33 bytes.

Return Value

Returns a pointer to string.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_ui64toa				
_ui64tow		+		

1.1.2.29.46 ultoa, _ultow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *ultoa(unsigned long value, char *string, int radix);
wchar_t *_ultow(unsigned long value, wchar_t *string, int radix);
```

Description

Converts an **unsigned long** to a string.

ultoa converts value to a null-terminated string and stores the result in string. value is an **unsigned long**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. ultoa performs no overflow checking, and if value is negative and radix equals 10, it does not set the minus sign.

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). ultoa can return up to 33 bytes.

Return Value

ultoa returns string.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main( void )
{
   unsigned long lnumber = 3123456789L;
   char string[25];
   ultoa(lnumber,string,10);
   printf("string = %s  unsigned long = %lu\n",string,lnumber);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ultoa		+		
_ultow		+		

1.1.2.29.47 wcstombs

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Description

Converts a **wchar_t** array into a multibyte string.

wcstombs converts the type **wchar_t** elements contained in pwcs into a multibyte character string s. The process terminates if either a null character or an invalid multibyte character is encountered.

No more than n bytes are modified. If n number of bytes are processed before a null character is reached, the array s is not null terminated.

The behavior of wcstombs is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If an invalid multibyte character is encountered, wcstombs returns (size_t) -1. Otherwise, the function returns the number of bytes modified, not including the terminating code, if any.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int x;
  char *pbuf = (char*)malloc( MB_CUR_MAX);
  wchar_t *pwcsEOL = L'\0';
  char *pwchi= L"Hi there!";

  printf (" Convert entire wchar string into a multibyte string:\n");
  x = wcstombs( pbuf, pwchi,MB_CUR_MAX);
  printf ("Character converted: %u\n", x);
  printf ("Multibyte string character: %1s\n\n",pbuf);
  printf (" Convert when target is NULL\n");
  x = wcstombs( pbuf, pwcsEOL, MB_CUR_MAX);
  printf ("Character converted: %u\n",x);
  printf ("Multibyte string: %1s\n\n",pbuf);

}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.48 wctomb

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
int wctomb(char *s, wchar_t wc);
```

Description

Converts **wchar_t** code to a multibyte character.

If s is not null, wctomb determines the number of bytes needed to represent the multibyte character corresponding to wc (including any change in shift state). The multibyte character is stored in s. At most MB_CUR_MAX characters are stored. If the value of wc is zero, wctomb is left in the initial state.

The behavior of wctomb is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If s is a NULL pointer, wctomb returns a nonzero value if multibyte character encodings do have state-dependent encodings, and a zero value if they do not.

If s is not a NULL pointer, wctomb returns -1 if the wc value does not represent a valid multibyte character. Otherwise, wctomb returns the number of bytes that are contained in the multibyte character corresponding to wc. In no case will the return value be greater than the value of MB_CUR_MAX macro.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
```

```
{
  int x;
  wchar_t wc = L'a';
  char *pmbNULL = NULL;
  char *pmb = (char *)malloc(sizeof( char ));
  printf (" Convert a wchar_t array into a multibyte string:\n");
  x = wctomb( pmb, wc);
  printf ("Character converted: %u\n", x);
  printf ("Multibyte string: %1s\n\n",pmb);
  printf (" Convert when target is NULL\n");
  x = wctomb( pmbNULL, wc);
  printf ("Character converted: %u\n",x);
  printf ("Multibyte stri ng: %1s\n\n",pmbNULL);

}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.29.49 _argc

Header File

stdlib.h

Syntax

```
extern int _argc;
```

Description

_argc has the same value as argc (passed to main) when the program starts. This variable holds the number of arguments passed to the program. The value includes the name of the program itself, so _argc and argc are always at least 1.

1.1.2.29.50 _argv, _wargv

Header File

stdlib.h

Syntax

```
extern char **_argv;
```

```
extern wchar_t ** _wargv
```

Description

_argv points to an array containing the original command-line arguments (the elements of argv[]) passed to main when the program starts.

_wargv is the Unicode version of _argv.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_argv		+		
_wargv		NT only		

1.1.2.29.51 _environ, _wenviron

Header File

stdlib.h

Syntax

```
extern char ** _environ;
```

```
extern wchar_t ** _wenviron
```

Description

_environ is an array of pointers to strings; it is used to access and alter the operating system environment variables. Each string is of the form:

```
envvar = varvalue
```

where envvar is the name of an environment variable (such as PATH), and varvalue is the string value to which envvar is set (such as C:\Utils;C:\MyPrograms). The string varvalue can be empty.

When a program begins execution, the operating system environment settings are passed directly to the program. Note that env, the third argument to main, is equal to the initial setting of _environ.

The _environ array can be accessed by getenv; however, the putenv function is the only routine that should be used to add, change or delete the _environ array entries. This is because modification can resize and relocate the process environment array, but _environ is automatically adjusted so that it always points to the array.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_environ		+		
_wenviron		NT only		

1.1.2.29.52 EXIT_xxxx #defines

Header File

stdlib.h

Description

Constants defining exit conditions for calls to the exit function.

Name	Meaning
EXIT_SUCCESS	Normal program termination
EXIT_FAILURE	Abnormal program termination

1.1.2.29.53 RAND_MAX #define

Header File

stdlib.h

Syntax

Description

Maximum value returned by rand function.

1.1.2.30 string.h

The following functions, macros, and classes are provided in `string.h`:

1.1.2.30.1 _ismbblead, _ismbbtrail

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _ismbblead(unsigned int c);
int _ismbbtrail(unsigned int c);
```

Description

_ismbblead and _ismbbtrail are used to test whether the argument c is the first or the second byte of a multibyte character.

_ismbblead and _ismbbtrail are affected by the code page in use. You can set the code page by using the _setlocale function.

Return Value

If c is in the lead byte of a multibyte character, _ismbblead returns true.

If c is in the trail byte of a multibyte character, _ismbbtrail returns a nonzero value.

1.1.2.30.2 _ismbclegal

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _ismbclegal(unsigned int c);
```

Description

_ismbclegal tests whether each byte of the c argument is in the code page that is currently in use.

Return Value

_ismbclegal returns a nonzero value if the argument c is a valid multibyte character on the current code page. Otherwise, the function returns zero.

1.1.2.30.3 _ismbslead, _ismbstrail

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _ismbslead(const unsigned char *s1, const unsigned char *s2);
int _ismbstrail(const unsigned char *s1, const unsigned char *s2);
```

Description

The _ismbslead and _ismbstrail functions test the s1 argument to determine whether the s2 argument is a pointer to the lead byte or the trail byte. The test is case-sensitive.

Return Value

The _ismbslead and _ismbstrail routines return -1 if s2 points to a lead byte or a trail byte, respectively. If the test is false, the routines return zero.

1.1.2.30.4 _mbbtype

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _mbbtype(unsigned char ch, int mode);
```

Description

The _mbbtype function inspects the multibyte argument, character ch, to determine whether it is a single-byte character, or whether ch is the leadbyte or trailing byte in a multibyte character. The _mbbtype function can determine whether ch is an invalid character.

Return Value

The value that _mbbtype returns is one of the following manifest constants, defined in mbctype.h. The return value depends on the value of ch and the test which you want performed on ch.

1.1.2.30.5 _mbccpy

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
void _mbccpy(unsigned char *dest, unsigned char *src);
```

Description

The _mbccpy function copies a multibyte character from src to dest. The _mbccpy function makes an implicit call to _ismbblead so that the src pointer references a lead byte. If src doesn't reference a lead byte, no copy is performed.

Return Value

None.

1.1.2.30.6 _mbsbtype

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _mbsbtype(const unsigned char *str, size_t nbyte);
```

Description

The nbyte argument specifies the number of bytes from the start of the zero-based string.

The _mbsbtype function inspects the argument str to determine whether the byte at the position specified by nbyte is a single-byte character, or whether it is the leadbyte or trailing byte in a multibyte character. The _mbsbtype function can determine whether the byte pointed at is an invalid character or a NULL byte.

Any invalid bytes in str before nbyte are ignored.

Return Value

The value that _mbsbtype returns is one of the following manifest constants, defined in mbctype.h.

1.1.2.30.7 _mbsnbcmp

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _mbsnbcmp(const unsigned char *s1, const unsigned char s2, size_t maxlen);
```

Description

_mbsnbcmp makes an case-sensitive comparison of s1 and s2 for no more than maxlen bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined maxlen bytes.

_mbsnbcmp is case sensitive.

_mbsnbcmp is not affected by locale.

_mbsnbcmp compares bytes based on the current multibyte code page.

Return Value

- _mbsnbcmp returns an integer value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

1.1.2.30.8 _mbsnbcoll, _mbsnbicoll

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _mbsnbcoll(const unsigned char *s1, const unsigned char *s2, maxlen);
int _mbsnbicoll(const unsigned char *s1, const unsigned char *s2, maxlen);
```

Description

_mbsnbicoll is the case-insensitive version of _mbsnbcoll.

These functions collate the strings specified by arguments s1 and s2. The collation order is determined by lexicographic order as specified by the current multibyte code page. At most, maxlen number of bytes are collated.

Note: **Note:** The lexicographic order is not always the same as the order of characters in the character set.

If the last byte in s1 or s2 is a leadbyte, it is not compared.

Return Value

- Each of these functions return an integer value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

On error, each of these functions returns _**NLSCMPERROR**.

1.1.2.30.9 _mbsnbcpy

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsnbcpy(unsigned char *dest, unsigned char *src, size_t maxlen);
```

Description

The _mbsnbcpy function copies at most maxlen number of characters from the src buffer to the dest buffer. The dest buffer is null terminated after the copy.

It is the user's responsibility to be sure that dest is large enough to allow the copy. An improper buffer size can result in memory corruption.

Return Value

The function returns dest.

1.1.2.30.10 _mbsnbicmp

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _mbsnbicmp(const unsigned char *s1, const unsigned char s2, size_t maxlen);
```

Description

_mbsnbicmp ignores case while making a comparison of s1 and s2 for no more than maxlen bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined maxlen bytes.

_mbsnbicmp is not case sensitive.

_mbsnbicmp is not affected by locale.

_mbsnbicmp compares bytes based on the current multibyte code page.

Return Value

- _mbsnbicmp returns an integer value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

1.1.2.30.11 _mbsnbset

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsnbset(unsigned char str, unsigned int ch, size_t maxlen);
```

Description

_mbsnbset sets at most maxlen number of bytes in the string str to the character ch. The argument ch can be a single or

multibyte character.

The function quits if the terminating null character is found before maxlen is reached. If ch is a multibyte character that cannot be accomodated at the end of str, the last character in str is set to a blank character.

Return Value

strset returns str.

1.1.2.30.12 _mbsninc, _strninc, _wcsninc

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsninc(const unsigned char *str, size_t num);
```

Description

The functions increment the character array str by num number of characters.

These functions should be accessed through the portable macro, _tcsninc, defined in tchar.h.

Return value

The functions return a pointer to the resized character string specified by the argument str.

1.1.2.30.13 _mbsnbcnt, _mbsnccnt, _strncnt, _wcsncnt

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t _mbsnbcnt(const unsigned char * str, size_t nmbc);
size_t _mbsnccnt(const unsigned char * str, size_t nbyte);
```

Description

If _MBCS is defined:

_mbsnbcnt is mapped to the portable macro _tcsnbcnt

_mbsnccnt is mapped to the portable macro _tcsnccnt If _UNICODE is defined:

both _mbsnbcnt and _mbsnccnt are mapped to the _wcsncnt macro

If neither _MBCS nor _UNICODE are defined.

_tcsnbcnt and _tcsnccnt are mapped to the _strncnt macro _strncnt is the single-byte version of these functions. _wcsncnt is the wide-character version of these functions.

_strncnt and _wcsncnt are available only for generic-text mappings. They should not be used directly.

_mbsnbcnt examines the first nmbc multibyte characters of the str argument. The function returns the number of bytes found in the those characters.

_mbsnccnt examines the first nmbc bytes of the str argument. The function returns the number of characters found in those bytes. If NULL is encountered in the second byte of a multibyte character, the whole character is considered NULL and will not be included in the return value.

Each of the functions ends its examination of the str argument if NULL is reached before the specified number of characters or bytes is examined.

If str has fewer than the specified number of characters or bytes, the function return the number of characters or bytes found in str.

Return Value

_mbsnbcnt returns the number of bytes found.

_mbsnccnt returns the number of characters found.

If nmbc or nbyte are less than zero, the functions return 0.

1.1.2.30.14 _mbsspnp, _strspnp, _wcsspnp

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsspnp(const unsigned char *s1, const unsigned char *s2);
```

Description

Each of these functions search for the first character in s1 that is not contained in s2.

Use the portable macro, _tcsspnp, defined in tchar.h, to access these functions.

Return Value

The functions return a pointer to the first character in s1 that is not found in the character set for s2.

If every character from s1 is found in s2, each of the functions return NULL.

1.1.2.30.15 stpcpy, _wstpcpy, _stpcpy

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
char *stpcpy(char *dest, const char *src);
```

```
wchar * _wcspcpy(wchar *dest, const wchar *src);
```

Description

Copies one string into another.

_stpcpy copies the string src to dest, stopping after the terminating null character of src has been reached.

Return Value

stpcpy returns a pointer to the terminating null character of dest.

If UNICODE is defined, _wcspcpy returns a pointer to the terminating null character of the **wchar_t** dest string.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";
   stpcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_stpcpy		+		
_wcspcpy		+		

1.1.2.30.16 strcat, _mbscat, wcscat

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strcat(char *dest, const char *src);
wchar_t *wcscat(wchar_t *dest, const wchar_t *src);
unsigned char *_mbscat(unsigned char *dest, const unsigned char *src);
```

Description

Appends one string to another.

strcat appends a copy of src to the end of dest. The length of the resulting string is strlen(dest) + strlen(src).

Return Value

strcat returns a pointer to the concatenated strings.

Example

```
#include <string.h>
#include <stdio.h>
```

```
int main(void)
{
   char destination[25];
   char *blank = " ", *c = "C++", *CodeGear = "CodeGear";
   strcpy(destination, CodeGear);
   strcat(destination, blank);
   strcat(destination, c);
   printf("%s\n", destination);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcat	+	+	+	+
_mbscat		+		
wcscat		+	+	+

1.1.2.30.17 strchr, _mbschr, wcschr

Header File

string.h

Category

Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines

Prototype

```
char *strchr(const char *s, int c);/* C only */
```

```
const char *strchr(const char *s, int c);// C++ only
```

```
char *strchr( char *s, int c);// C++ only
```

```
wchar_t *wcschr(const wchar_t *s, int c);
```

```
unsigned char * _mbschr(const unsigned char *s, unsigned int c);
```

Description

Scans a string for the first occurrence of a given character.

strchr scans a string in the forward direction, looking for a specific character. strchr finds the first occurrence of the character c in the string s. The null-terminator is considered to be part of the string.

For example:

```
strchr(strs,0)
```

returns a pointer to the terminating null character of the string strs.

Return Value

strchr returns a pointer to the first occurrence of the character c in s; if c does not occur in s, strchr returns null.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char string[15];
```

```
   char *ptr, c = 'r';
   strcpy(string, "This is a string");
   ptr = strchr(string, c);
   if (ptr)
      printf("The character %c is at position: %d\n", c, ptr-string);
   else
      printf("The character was not found\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strchr	+	+	+	+
_mbschr		+		
wcschr		+	+	+

1.1.2.30.18 strcmp, _mbscmp, wcscmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
int strcmp(const char *s1, const char *s2);
```

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

```
int _mbscmp(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares one string to another.

strcmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
int ptr;
ptr = strcmp(buf2, buf1);
if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");
```

```
   ptr = strcmp(buf2, buf3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcmp	+	+	+	+
_mbscmp		+		
wcscmp		+	+	+

1.1.2.30.19 strcmpi

Header File

string.h, wchar.h

Category

Memory and String Manipulation Routines

Prototype

```
int strcmpi(const char *s1, const char *s2);
```

```
int _wcscmpi(const wchar_t *s1, const wchar_t *s2);
```

Description

Compares one string to another, without case sensitivity.

strcmpi performs an unsigned comparison of s1 to s2, without case sensitivity (same as stricmp--implemented as a macro).

It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

The routine strcmpi is the same as stricmp. strcmpi is implemented through a macro in string.h and translates calls from strcmpi to stricmp. Therefore, in order to use strcmpi, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
/* strncmpi example */
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBB", *buf2 = "bbb";
   int ptr;
   ptr = strcmpi(buf2, buf1);
   if (ptr > 0)
```

```
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcmpi		+		
_wcscmpi		+		

1.1.2.30.20 strcoll,_stricoll, _mbscoll, _mbsicoll, wcscoll, _wcsicoll

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int strcoll(const char *s1, const char *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int _stricoll(const char *s1, const char *s2);
int _wcsicoll(const wchar_t *s1, wconst_t char *s2);
int _mbscoll(const unsigned char *s1, const unsigned char *s2);
int _mbsicoll(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares two strings.

strcoll compares the string pointed to by s1 to the string pointed to by s2, according to the current locale's LC_COLLATE category.

_stricoll performs like strcoll but is not case sensitive.

Note: Note

_stricoll does not compare string according to the current locale's LC_COLLATE category. _stricoll gives you a stricmp. The required collation is obtained by calling _lstricoll, or just plain stricoll (which maps to _lstricoll).

The real collation (_lstricoll) returns -1, 0 or 1, whereas _stricoll does a codepoint comparison, and returns < 0, 0 or > 0.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *two = "International";
   char *one = "Borland";
   int check;
   check = strcoll(one, two);
   if (check == 0)
      printf("The strings are equal\n");
   if (check <  0)
      printf("%s comes before %s\n", one, two);
   if (check >  0)
      printf("%s comes before %s\n", two, one);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcoll	+	+	+	+
_stricoll		+		
_mbscoll		+		
_mbsicoll		+		
wcscoll		+	+	+
_wcsicoll		+		

1.1.2.30.21 strcpy

Header File

string.h, wchar.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

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

```
wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);
```

```
unsigned char *_mbscpy(unsigned char *dest, const unsigned char *src);
```

Description

Copies one string into another.

Copies string src to dest, stopping after the terminating null character has been moved.

Return Value

strcpy returns dest.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
```

```
   char string[10];
   char *str1 = "abcdefghi";
   strcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcpy	+	+	+	+
_mbscpy		+		
wcscpy		+	+	+

1.1.2.30.22 strcspn, _mbscspn, wcscspn

Header File

string.h, wchar.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t strcspn(const char *s1, const char *s2);
```

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

```
size_t _mbscspn(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the initial segment not containing any subset of a given set of characters.

The strcspn functions search s1 until any one of the characters contained in s2 is found. The number of characters which were read in s1 is the return value. The string termination character is not counted. Neither string is altered during the search.

Return Value

strcspn returns the length of the initial segment of string s1 that consists entirely of characters not from string s2.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
   char *string1 = "1234567890";
   char *string2 = "747DC8";
   int length;
   length = strcspn(string1, string2);
   printf("Character where strings intersect is at position %d\n",
          length);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcspn	+	+	+	+
_mbscspn		+		
wcscspn		+	+	+

1.1.2.30.23 _strdec, mbsdec, _wcsdec

Header File

mbstring.h, tchar.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsdec(const unsigned char *s, const unsigned char *p);
inline char *_strdec(const char * s1, const char * s2) { return (char *)(s1,(s2-1));
// From tchar.h
#define _tcsdec _strdec
#define _tcsdec _wcsdec
```

Description

_mbsdec returns a pointer p indicating the character in string s back to 1 byte backward. If there are no more characters before string p (it is the same position as s), _mbsdec returns a **null** pointer.

_strdec is the single-byte version of this function.

_wcsdec is the wide-character version of this function.

Return Value

Returns a pointer back to 1 byte, or **null** pointer if there is no character before p.

1.1.2.30.24 strdup, _mbsdup, _wcsdup

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strdup(const char *s);
wchar_t *_wcsdup(const wchar_t *s);
unsigned char *_mbsdup(const wchar_t *s);
```

Description

Copies a string into a newly created location.

strdup makes a duplicate of string s, obtaining space with a call to malloc. The allocated space is (strlen(s) + 1) bytes long. The user is responsible for freeing the space allocated by strdup when it is no longer needed.

Return Value

strdup returns a pointer to the storage location containing the duplicated string, or returns null if space could not be allocated.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
   char *dup_str, *string = "abcde";
   dup_str = strdup(string);
   printf("%s\n", dup_str);
   free(dup_str);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strdup		+		
_mbsdup		+		
_wcsdup		+		

1.1.2.30.25 strerror

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strerror(int errnum);
```

Description

Returns a pointer to an error message string.

strerror takes an **int** parameter errnum, an error number, and returns a pointer to an error message string associated with errnum.

Return Value

strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to strerror.

Example

```
#include <stdio.h>
#include <errno.h>
int main(void)
{
  char *buffer;
  buffer = strerror(errno);
```

```
  printf("Error: %s\n", buffer);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.30.26 stricmp, _mbsicmp, _wcsicmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int stricmp(const char *s1, const char *s2);
```

```
int _wcsicmp(const wchar_t *s1, const wchar_t *s2);
```

```
int _mbsicmp(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares one string to another, without case sensitivity.

stricmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

The routines stricmp and strcmpi are the same; strcmpi is implemented through a macro in string.h that translates calls from strcmpi to stricmp. Therefore, in order to use stricmp, you must include the header file string.h for the macro to be available.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBB", *buf2 = "bbb";
   int ptr;
   ptr = stricmp(buf2, buf1);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
```

```
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
stricmp		+	+	+
_mbsicmp		+		
_wcsicmp		+		

1.1.2.30.27 _strinc, mbsinc, _wcsinc

Header File

mbstring.h, tchar.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsinc(const unsigned char *p);
// From tchar.h
#define _tcsinc _strinc
#define _tcsinc _wcsinc
inline char * strinc(const char * s) { return (char *)(s+1); }
```

Description

_mbsinc increments a string pointer by one byte.

_strdec is the single-byte version of this function.

_wcsdec is the wide-character version of this function.

Return Value

Returns a pointer that is forwarded by 1 byte.

1.1.2.30.28 strlwr, _mbslwr, _wcslwr

Header File

string.h, mbstring.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
char *strlwr(char *s);
wchar_t *_wcslwr(wchar_t *s);
unsigned char *_mbslwr(unsigned char *s);
```

Description

Converts uppercase letters in a string to lowercase.

strlwr converts uppercase letters in string s to lowercase according to the current locale's LC_CTYPE category. For the C locale, the conversion is from uppercase letters (A to Z) to lowercase letters (a to z). No other characters are changed.

Return Value

strlwr returns a pointer to the string s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "CodeGear";
   printf("string prior to strlwr: %s\n", string);
   strlwr(string);
   printf("string after strlwr:    %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strlwr		+		
_mbslwr		+		
_wcslwr		+		

1.1.2.30.29 strncat

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strncat(char *dest, const char *src, size_t maxlen);
wchar_t *wcsncat(wchar_t *dest, const wchar_t *src, size_t maxlen);
unsigned char *_mbsncat(unsigned char *dest, const unsigned char *src, size_t maxlen);
unsigned char *_mbsnbcat(unsigned char *__dest, const unsigned char *__src, _SIZE_T __maxlen);
```

Description

Appends a portion of one string to another.

strncat copies at most maxlen characters of src to the end of dest and then appends a null character. The maximum length of the resulting string is strlen(dest) + maxlen.

For _mbsnbcat, if the second byte of 2-bytes character is null, the first byte of this character is regarded as null.

These four functions behave identically and differ only with respect to the type of arguments and return types.

Return Value

strncat returns dest.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char destination[25];
   char *source = " States";
   strcpy(destination, "United");
   strncat(destination, source, 7);
   printf("%s\n", destination);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncat	+	+	+	+
_mbsncat		+		
_mbsnbcat		+		
_wcsncat		+		

1.1.2.30.30 strncmp, _mbsncmp, wcsncmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
int strncmp(const char *s1, const char *s2, size_t maxlen);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);
int _mbsncmp(const unsigned char *s1, const unsigned char *s2, size_t maxlen);
#define _mbccmp(__s1, __s2) _mbsncmp((__s1),(__s2),1)
```

Description

Compares a portion of one string to a portion of another.

strncmp makes the same unsigned comparison as strcmp, but looks at no more than maxlen characters. It starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until it has examined maxlen characters.

Return Value

- strncmp returns an int value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

Example

```
#include <string.h>
```

```
#include <stdio.h>
int  main(void)
{
   char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
   int ptr;
   ptr = strncmp(buf2,buf1,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");
   ptr = strncmp(buf2,buf3,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");
   return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncmp	+	+	+	+
_mbsncmp		+		
_mbccmp		+		
wcsncmp		+	+	+

1.1.2.30.31 strncoll, strnicoll, _mbsncoll, _mbsnicoll, _wcsncoll, _wcsnicoll

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _strncoll(const char *s1, const char *s2, size_t n);
int _wcsncoll(const wchar_t *s1, const wchar_t *s2, size_t n);
int _strnicoll(const char *s1, const char *s2, size_t n);
int _wcsnicoll(const wchar_t *s1, const wchar_t *s2, size_t n);
int _mbsncoll(const unsigned char *s1, const unsigned char *s2, size_t n);
int _mbsnicoll(const unsigned char *s1, const unsigned char *s2, size_t n);
```

Description

_strncoll compares n number of elements from the string pointed to by s1 to the string pointed to by s2, according to the current locale's LC_COLLATE category.

_strnicoll performs like _strncoll but is not case sensitive.

Return Value

less than s2	< 0
the same as s2	== 0

greater than s2	> 0

1.1.2.30.32 strncmpi, wcsncmpi

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
int strncmpi(const char *s1, const char *s2, size_t n);
```

```
int wcsncmpi(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

Compares a portion of one string to a portion of another, without case sensitivity.

strncmpi performs a signed comparison of s1 to s2, for a maximum length of n bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until n characters have been examined. The comparison is not case sensitive. (strncmpi is the same as strnicmp--implemented as a macro). It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

The routines strnicmp and strncmpi are the same; strncmpi is implemented through a macro in string.h that translates calls from strncmpi to strnicmp. Therefore, in order to use strncmpi, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBBccc", *buf2 = "bbbccc";
   int ptr;
   ptr = strncmpi(buf2,buf1,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncmpi		+		

wcsncmpi		+		

1.1.2.30.33 strncpy, _mbsncpy, wcsncpy

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strncpy(char *dest, const char *src, size_t maxlen);
wchar_t *wcsncpy(wchar_t *dest, const wchar_t *src, size_t maxlen);
unsigned char *_mbsncpy(unsigned char *dest, const unsigned char *src, size_t maxlen);
```

Description

Copies a given number of bytes from one string into another, truncating or padding as necessary.

strncpy copies up to maxlen characters from src into dest, truncating or null-padding dest. The target string, dest, might not be null-terminated if the length of src is maxlen or more.

Return Value

strncpy returns dest.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";
   strncpy(string, str1, 3);
   string[3] = '\0';
   printf("%s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncpy	+	+	+	+
_mbsncpy		+		
wcsncpy		+	+	+

1.1.2.30.34 _strnextc,_mbsnextc,_wcsnextc

Header File

tchar.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned int _strnextc(const char *str);
```

```
unsigned int _mbsnextc (const unsigned char *str);
```

Description

These routines should be accessed by using the portable _tcsnextc function. The functions inspect the current character in str. The pointer to str is not advanced.

Return Value

The functions return the integer value of the character pointed to by str.

Example

```
#include <tchar.h>
#include <stdio.h>
int main()
{
   unsigned int retval = 0;
   const unsigned char *string = "ABC";
   retval = _strnextc(string);
   printf("The starting character:%c", retval);
   retval = _strnextc(++string);
   printf("\nThe next character:%c", retval);
   return 0;
}
/***
The starting character:A
The next character:B
***/
```

1.1.2.30.35 strnicmp, _mbsnicmp, _wcsnicmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int strnicmp(const char *s1, const char *s2, size_t maxlen);
```

```
int _wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);
```

```
int _mbsnicmp(const unsigned char *s1, const unsigned char *s2, size_t maxlen);
```

Description

Compares a portion of one string to a portion of another, without case sensitivity.

strnicmp performs a signed comparison of s1 to s2, for a maximum length of maxlen bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBBccc", *buf2 = "bbbccc";
   int ptr;
   ptr = strnicmp(buf2, buf1, 3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strnicmp		+		
_mbsnicmp		+		
_wcsnicmp		+		

1.1.2.30.36 strnset, _mbsnset, _wcsnset

Header File

string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strnset(char *s, int ch, size_t n);
```

```
wchar_t *_wcsnset(wchar_t *s, wchar_t ch, size_t n);
```

```
unsigned char *_mbsnset(unsigned char *s, unsigned int ch, size_t n);
```

Description

Sets a specified number of characters in a string to a given character.

strnset copies the character ch into the first n bytes of the string s. If n > strlen(s), then strlen(s) replaces n. It stops when n characters have been set, or when a null character is found.

Return Value

Each of these functions return s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "abcdefghijklmnopqrstuvwxyz";
   char letter = 'x';
   printf("string before strnset: %s\n", string);
   strnset(string, letter, 13);
   printf("string after  strnset: %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strnset		+		
_mbsnset		+		
_wcsnset		+		

1.1.2.30.37 strpbrk, _mbspbrk, wcspbrk

Header File

string.h, mbstring.h

Category

C++ Prototyped Routines, Memory and String Manipulation Routines

Prototype

```
char *strpbrk(const char *s1, const char *s2); /* C only */
const char *strpbrk(const char *s1, const char *s2); // C++ only
char *strpbrk(char *s1, const char *s2); // C++ only
wchar_t * wcspbrk(const wchar_t *s1, const wchar_t *s2);
unsigned char *_mbspbrk(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the first occurrence of any character from a given set.

strpbrk scans a string, s1, for the first occurrence of any character appearing in s2.

Return Value

strpbrk returns a pointer to the first occurrence of any of the characters in s2. If none of the s2 characters occur in s1, strpbrk returns null.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "onm";
   char *ptr;
   ptr = strpbrk(string1, string2);
   if (ptr)
```

```
      printf("strpbrk found first character: %c\n", *ptr);
   else
      printf("strpbrk didn't find character in set\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strpbrk	+	+	+	+
_mbspbrk		+		
wcspbrk		+	+	+

1.1.2.30.38 strrchr, _mbsrchr, wcsrchr

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines

Prototype

```
char *strrchr(const char *s, int c); /* C only */
const char *strrchr(const char *s, int c); // C++ only
char *strrchr(char *s, int c); // C++ only
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
unsigned char * _mbsrchr(const unsigned char *s, unsigned int c);
```

Description

Scans a string for the last occurrence of a given character.

strrchr scans a string in the reverse direction, looking for a specific character. strrchr finds the last occurrence of the character c in the string s. The null-terminator is considered to be part of the string.

Return Value

strrchr returns a pointer to the last occurrence of the character c. If c does not occur in s, strrchr returns null.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char string[15];
   char *ptr, c = 'r';
   strcpy(string, "This is a string");
   ptr = strrchr(string, c);
   if (ptr)
      printf("The character %c is at position: %d\n", c, ptr-string);
   else
      printf("The character was not found\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strrchr	+	+	+	+
_mbsrchr		+		
wcsrchr		+	+	+

1.1.2.30.39 strrev, _mbsrev, _wcsrev

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strrev(char *s);
```

```
wchar_t *_wcsrev(wchar_t *s);
```

```
unsigned char *_mbsrev(unsigned char *s);
```

Description

Reverses a string.

strrev changes all characters in a string to reverse order, except the terminating null character. (For example, it would change string\0 to gnirts\0.)

Return Value

strrev returns a pointer to the reversed string.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *forward = "string";
   printf("Before strrev(): %s\n", forward);
   strrev(forward);
   printf("After strrev():  %s\n", forward);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strrev		+		
_mbsrev		+		
_wcsrev		+		

1.1.2.30.40 strset, _mbsset, _wcsset

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strset(char *s, int ch);
```

```
wchar_t *_wcsset(wchar_t *s, wchar_t ch);
```

```
unsigned char *_mbsset(unsigned char *s, unsigned int ch);
```

Description

Sets all characters in a string to a given character.

strset sets all characters in the string s to the character ch. It quits when the terminating null character is found.

Return Value

strset returns s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10] = "123456789";
   char symbol = 'c';
   printf("Before strset(): %s\n", string);
   strset(string, symbol);
   printf("After strset():  %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strset		+		
_mbsset		+		
_wcsset		+		

1.1.2.30.41 strspn, _mbsspn, wcsspn

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t strspn(const char *s1, const char *s2);
```

```
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

```
size_t _mbsspn(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the first segment that is a subset of a given set of characters.

strspn finds the initial segment of string s1 that consists entirely of characters from string s2.

Return Value

strspn returns the length of the initial segment of s1 that consists entirely of characters from s2.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
   char *string1 = "1234567890";
   char *string2 = "123DC8";
   int length;
   length = strspn(string1, string2);
   printf("Character where strings differ is at position %d\n", length);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strspn	+	+	+	+
_mbsspn		+		
wcsspn		+	+	+

1.1.2.30.42 strstr, _mbsstr, wcsstr

Header File

string.h

Category

C++ Prototyped Routines, Memory and String Manipulation Routines

Prototype

```
char *strstr(const char *s1, const char *s2); /* C only */
const char *strstr(const char *s1, const char *s2); // C++ only
char *strstr(char *s1, const char *s2); // C++ only
wchar_t * wcsstr(const wchar_t *s1, const wchar_t *s2);
unsigned char * _mbsstr(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the occurrence of a given substring.

strstr scans s1 for the first occurrence of the substring s2.

Return Value

strstr returns a pointer to the element in s1, where s2 begins (points to s2 in s1). If s2 does not occur in s1, strstr returns null.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *str1 = "CodeGear", *str2 = "nation", *ptr;
   ptr = strstr(str1, str2);
   printf("The substring is: %s\n", ptr);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strstr	+	+	+	+
_mbsstr		+		
wcsstr		+	+	+

1.1.2.30.43 strtok, _mbstok, wcstok

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strtok(char *s1, const char *s2);
```

```
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2);
```

```
unsigned char *_mbstok(unsigned char *s1, const unsigned char *s2);
```

Description

Searches one string for tokens, which are separated by delimiters defined in a second string.

strtok considers the string s1 to consist of a sequence of zero or more text tokens, separated by spans of one or more characters from the separator string s2.

The first call to strtok returns a pointer to the first character of the first token in s1 and writes a null character into s1 immediately following the returned token. Subsequent calls with null for the first argument will work through the string s1 in this way, until no tokens remain.

The separator string, s2, can be different from call to call.

Note: Calls to strtok cannot be nested with a function call that also uses strtok. Doing so will causes an endless loop.

Return Value

strtok returns a pointer to the token found in s1. A NULL pointer is returned when there are no more tokens.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char input[16] = "abc,d";
   char *p;
```

```
    /* strtok places a NULL terminator
    in front of the token, if found */
    p = strtok(input, ",");
    if (p)   printf("%s\n", p);
    /* A second call to strtok using a NULL
    as the first parameter returns a pointer
    to the character following the token  */
    p = strtok(NULL, ",");
    if (p)   printf("%s\n", p);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtok	+	+	+	+
_mbstok		+		
wcstok		+	+	+

1.1.2.30.44 strupr, _mbsupr, _wcsupr

Header File

string.h, mbstring.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
char *strupr(char *s);

wchar_t *_wcsupr(wchar_t *s);

unsigned char *_mbsupr(unsigned char *s);
```

Description

Converts lowercase letters in a string to uppercase.

strupr converts lowercase letters in string s to uppercase according to the current locale's LC_CTYPE category. For the default C locale, the conversion is from lowercase letters (a to z) to uppercase letters (A to Z). No other characters are changed.

Return Value

strupr returns s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "abcdefghijklmnopqrstuvwxyz", *ptr;
   /* converts string to upper case characters */
   ptr = strupr(string);
   printf("%s\n", ptr);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strupr		+		
_mbsupr		+		
_wcsupr		+		

1.1.2.30.45 strxfrm, wcsxfrm

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t strxfrm(char *target, const char *source, size_t n);
```

```
size_t wcsxfrm(wchar_t *target, const wchar_t *source, size_t n);
```

Description

Transforms a portion of a string to a specified collation.

strxfrm transforms the string pointed to by source into the string target for no more than n characters. The transformation is such that if the strcmp function is applied to the resulting strings, its return corresponds with the return values of the strcoll function.

No more than n characters, including the terminating null character, are copied to target.

strxfrm transforms a character string into a special string according to the current locale's LC_COLLATE category. The special string that is built can be compared with another of the same type, byte for byte, to achieve a locale-correct collation result. These special strings, which can be thought of as keys or tokenized strings, are not compatible across the different locales.

The tokens in the tokenized strings are built from the collation weights used by strcoll from the active locale's collation tables.

Processing stops only after all levels have been processed for the character string or the length of the tokenized string is equal to the maxlen parameter.

All redundant tokens are removed from each level's set of tokens.

The tokenized string buffer must be large enough to contain the resulting tokenized string. The length of this buffer depends on the size of the character string, the number of collation levels, the rules for each level and whether there are any special characters in the character string. Certain special characters can cause extra character processing of the string resulting in more space requirements. For example, the French character "oe" will take double the space for itself because in some locales, it expands to collation weights for each level. Substrings that have substitutions will also cause extra space requirements.

There is no safe formula to determine the required string buffer size, but at least (levels * string length) are required.

Return Value

Number of characters copied not including the terminating null character. If the value returned is greater than or equal to n, the content of target is indeterminate.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
```

```
{
   char *target;
   char *source = "Frank Borland";
   int length;
   /* allocate space for the target string */
   target = (char *) calloc(80, sizeof(char));
   /* copy the source over to the target and get the length */
   length = strxfrm(target, source, 80);
   /* print out the results */
   printf("%s has the length %d\n", target, length);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strxfrm	+	+	+	+
wcsxfrm		+	+	+

1.1.2.31 sys\stat.h

The following functions, macros, and classes are provided in `sys\stat.h`:

1.1.2.31.1 fstat, stat, _wstat

Header File

sys\stat.h

Category

Input/output Routines

Prototype

```
int fstat(int handle, struct stat *statbuf);
```

```
int stat(const char *path, struct stat *statbuf);
```

```
int _wstat(const wchar_t *path, struct stat *statbuf);
```

Description

Gets open file information.

fstat stores information in the stat structure about the file or directory associated with handle.

stat stores information about a given file or directory in the stat structure. The name of the file is path.

statbuf points to the stat structure (defined in sys\stat.h). That structure contains the following fields:

st_mode	Bit mask giving information about the file's mode
st_dev	Drive number of disk containing the file or file handle if the file is on a device
st_rdev	Same as st_dev
st_nlink	Set to the integer constant 1
st_size	Size of the file in bytes
st_atime	Most recent access (Windows) or last time modified (DOS)

st_mtime	Same as st_atime
st_ctime	Same as st_atime

The stat structure contains three more fields not mentioned here. They contain values that are meaningful only in UNIX.

The st_mode bit mask that gives information about the mode of the open file includes the following bits:

One of the following bits will be set:

S_IFCHR	If handle refers to a device.
S_IFREG	If an ordinary file is referred to by handle.

One or both of the following bits will be set:

S_IWRITE	If user has permission to write to file.
S_IREAD	If user has permission to read to file.

The HPFS and NTFS file-management systems make the following distinctions:

st_atime	Most recent access
st_mtime	Most recent modify
st_ctime	Creation time

Return Value

fstat and stat return 0 if they successfully retrieved the information about the open file.

On error (failure to get the information) these functions return -1 and set the global variable errno to

EBADF	Bad file handle

Example

```
#include <sys\stat.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
   struct stat statbuf;
   FILE *stream;
   /* open a file for update */
   if ((stream = fopen("DUMMY.FIL", "w+"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return(1);
   }
   fprintf(stream, "This is a test");
   fflush(stream);
   /* get information about the file */
   fstat(fileno(stream), &statbuf);
   fclose(stream);
   /* display the information returned */
   if (statbuf.st_mode & S_IFCHR)
      printf("Handle refers to a device.\n");
   if (statbuf.st_mode & S_IFREG)
      printf("Handle refers to an ordinary file.\n");
   if (statbuf.st_mode & S_IREAD)
```

```
      printf("User has read permission on file.\n");
   if (statbuf.st_mode & S_IWRITE)
      printf("User has write permission on file.\n");
   printf("Drive letter of file: %c\n", 'A'+statbuf.st_dev);
   printf("Size of file in bytes: %ld\n", statbuf.st_size);
   printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));
   return 0;
}
```

1.1.2.31.2 _stati64, _tstati64, stati64, _wstati64

Header File

sys\stat.h, tchar.h

Category

Directory Control Routines

Prototype

```
int stati64(const char *pathname, struct stati64 *buff);

int _stati64(const char *__path, struct stati64 *__statbuf);

int _wstati64(const wchar_t *pathname, struct stati64 *buff);

// From tchar.h

#define _tstati64 _stati64
```

Description

Gather statistics about the file named by *pathP and place them in the buffer *bufP.

The statistics fields are set thus:

st_devset to -1 if S_IFCHR, else set to drive holding the file.

st_ino0

st_modeUnix-style bit-set for file access rights

st_nlink1

st_uid0

st_gid0

st_rdevsame as st_dev

st_sizefile size (0 if S_IFDIR or S_IFCHR)

st_atimetime file last changed (seconds since 1970)

st_mtimesame as st_atime

st_ctimesame as st_atime

The file access rights bit-set may contain S_IFCHR, S_IFDIR, S_IFREG, S_IREAD, S_IWRITE, or S_IEXEC.

If the name is for a device, the time fields will be zero and the size field is undefined.

Return Value

The return value is 0 if the call was successful, otherwise -1 is returned and errno contains the reason. The buffer is not touched unless the call is successful.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.31.3 S_Ixxxx #defines

Header File

sys\stat.h

Description

Definitions used for file status and directory functions.

Name	Meaning
S_IFMT	File type mask
S_IFDIR	Directory
S_IFIFO	FIFO special
S_IFCHR	Character special
S_IFBLK	Block special
S_IFREG	Regular file
S_IREAD	Owner can read
S_IWRITE	Owner can write
S_IEXEC	Owner can execute

1.1.2.32 sys\timeb.h

The following functions, macros, and classes are provided in `sys\timeb.h`:

1.1.2.32.1 ftime

Header File

sys\timeb.h

Category

Time and Date Routines

Prototype

```
void ftime(struct timeb *buf)
```

Description

Stores current time in timeb structure.

On UNIX platforms ftime is available only on System V systems.

ftime determines the current time and fills in the fields in the timeb structure pointed to by buf. The timeb structure contains four fields: time, millitm, _timezone, and dstflag:

```
struct timeb {
long time ;
short millitm ;
short _timezone ;
short dstflag ;
};
```

- timeprovides the time in seconds since 00:00:00 Greenwich mean time (GMT) January 1 1970.
- millitmis the fractional part of a second in milliseconds.
- _timezoneis the difference in minutes between GMT and the local time. This value is computed going west from GMT. ftime gets this field from the global variable _timezone which is set by tzset.
- dstflagis set to nonzero if daylight saving time is taken into account during time calculations.

Note: ftime calls tzset. Therefore it isn't necessary to call tzset explicitly when you use ftime.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys\timeb.h>
/* pacific standard & daylight savings */
char *tzstr = "TZ=PST8PDT";
int main(void)
{
   struct timeb t;
   putenv(tzstr);
   tzset();
   ftime(&t);
   printf("Seconds since 1/1/1970 GMT: %ld\n", t.time);
   printf("Thousandths of a second: %d\n", t.millitm);
   printf("Difference between local time and GMT: %d\n", t._timezone);
   printf("Daylight savings in effect (1) not (0): %d\n", t.dstflag);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.33 sys\types.h

The following functions, macros, and classes are provided in `sys\types.h`:

1.1.2.33.1 time_t

Header File

sys\types.h

time.h

Syntax

```
typedef long time_t;
```

Description

Defines the value used by the time functions declared in time.h.

1.1.2.34 time.h

The following functions, macros, and classes are provided in `time.h`:

1.1.2.34.1 asctime

Header File

time.h

Category

Time and Date Routines

Prototype

```
char *asctime(const struct tm *tblock);
```

```
wchar_t *_wasctime(const struct tm *tblock);
```

Description

asctime converts date and time to ASCII.

_wasctime converts date and time to a **wchar_t** string.

asctime and _wasctime convert a time stored as a structure to a 26 (wide) character string in the following form:

Mon Nov 21 11:31:54 1983\n\0

All the fields have a constant width. The output string day-of-the-week and month correspond to the following:

tm parameterValid value rangeOutput

tm.mon (month)0-110=Jan, 1=Feb, and so on

tm.day (day-of-the-week)0-60=Sun, 1=Mon, and so on

Return Value

asctime and _wasctime return a pointer to the (wide) character string containing the date and time. This string is a static which is overwritten with each call. asctime converts a time stored as a structure in *tblock to a 26-character string of the same form as the ctime string:

```
Sun Sep 16 01:03:52 1973\n\0
```

Example

```
#include <string.h>
#include <time.h>
#include <stdio.h>
int main(void)
{
```

```
  struct tm t;
  char str[80];
  /* sample loading of tm structure  */
  t.tm_sec    = 1;  /* Seconds */
  t.tm_min    = 30; /* Minutes */
  t.tm_hour   = 9;  /* Hour */
  t.tm_mday   = 22; /* Day of the Month  */
  t.tm_mon    = 11; /* Month */
  t.tm_year   = 56; /* Year - does not include century */
  t.tm_wday   = 4;  /* Day of the week  */
  t.tm_yday   = 0;  /* Does not show in asctime */
  t.tm_isdst  = 0;  /* Is Daylight SavTime; does not show in asctime */
  /* converts structure to null terminated string */
  strcpy(str, asctime(&t));
  printf("%s\n", str);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
asctime	+	+	+	+
_wasctime		+		

1.1.2.34.2 clock

Header File

time.h

Category

Time and Date Routines

Prototype

```
clock_t clock(void);
```

Description

Determines processor time.

clock can be used to determine the time interval between two events. To determine the time in seconds, the value returned by clock should be divided by the value of the macro CLK_TCK.

Return Value

On success, clock returns the processor time elapsed since the beginning of the program invocation.

On error (if the processor time is not available or its value cannot be represented), clock returns -1.

Example

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
int main(void)
{
   clock_t start, end;
   start = clock();
   delay(2000);
   end = clock();
   printf("The time was: %f\n", (end - start) / CLK_TCK);
```

```
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.3 ctime, _wctime

Header File

time.h

Category

Time and Date Routines

Prototype

```
char *ctime(const time_t *time);
```

```
wchar_t *_wctime(const time_t *time);
```

Description

Converts date and time to a string.

ctime converts a time value pointed to by time (the value returned by the function time) into a 26-character string in the following form, terminating with a newline character and a null character:

```
Mon Nov 21 11:31:54 1983\n\0
```

All the fields have constant width.

The global long variable _timezone contains the difference in seconds between GMT and local standard time (in PST, _timezone is 8*60*60). The global variable _daylight is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. It is set to 1 if the daylight savings time conversion should be applied.. These variables are set by the tzset function, not by the user program directly.

Return Value

ctime returns a pointer to the character string containing the date and time. The return value points to static data that is overwritten with each call to ctime.

Example

```
#include <stdio.h>
#include <time.h>
int main(void)
{
  time_t t;
  time(&t);
  printf("Today's date and time: %s\n", ctime(&t));
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ctime	+	+	+	+
_wctime		+		

1.1.2.34.4 difftime

Header File

time.h

Category

Time and Date Routines

Prototype

```
double difftime(time_t time2, time_t time1);
```

Description

Computes the difference between two times.

difftime calculates the elapsed time in seconds, from time1 to time2.

Return Value

difftime returns the result of its calculation as a **double**.

Example

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>
int main(void)
{
   time_t first, second;
   clrscr();
   first = time(NULL);  /* Gets system
                           time */
   delay(2000);         /* Waits 2 secs */
   second = time(NULL); /* Gets system time
                           again */
   printf("The difference is: %f seconds\n",difftime(second,first));
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.5 gmtime

Header File

time.h

Category

Time and Date Routines

Prototype

```
struct tm *gmtime(const time_t *timer);
```

Description

Converts date and time to Greenwich mean time (GMT).

gmtime accepts the address of a value returned by time and returns a pointer to the structure of type tm containing the time elements. gmtime converts directly to GMT.

The global long variable _timezone should be set to the difference in seconds between GMT and local standard time (in PST _timezone is 8 x 60 x 60). The global variable _daylight should be set to nonzero only if the standard U.S. daylight saving time conversion should be applied.

This is the tm structure declaration from the time.h header file:

```
struct tm {
int tm_sec; /* Seconds */
int tm_min; /* Minutes */
int tm_hour; /* Hour (0 - 23) */
int tm_mday; /* Day of month (1 - 31) */
int tm_mon; /* Month (0 - 11) */
int tm_year; /* Year (calendar year minus 1900) */
int tm_wday; /* Weekday (0 - 6; Sunday is 0) */
int tm_yday; /* Day of year (0 -365) */
int tm_isdst; /* Nonzero if daylight saving time is in effect. */
};
```


These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year - 1900, day of year (0 to 365), and a flag that is nonzero if daylight saving time is in effect.

Return Value

gmtime returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/* Pacific Standard Time & Daylight Savings */
char *tzstr = "TZ=PST8PDT";
int main(void)
{
   time_t t;
   struct tm *gmt, *area;
   putenv(tzstr);
   tzset();
   t = time(NULL);
   area = localtime(&t);
   printf("Local time is: %s", asctime(area));
   gmt = gmtime(&t);
   printf("GMT is:        %s", asctime(gmt));
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.6 localtime

Header File

time.h

Category

Time and Date Routines

Prototype

```
struct tm *localtime(const time_t *timer);
```

Description

Converts date and time to a structure.

localtime accepts the address of a value returned by time and returns a pointer to the structure of type tm containing the time elements. It corrects for the time zone and possible daylight saving time.

The global long variable _timezone contains the difference in seconds between GMT and local standard time (in PST, _timezone is 8 x 60 x 60). The global variable _daylight is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. It is set to 1 if the daylight savings time conversion should be applied. These values are set by tzset, not by the user program directly.

This is the **tm** structure declaration from the time.h header file:

```
struct tm {
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year - 1900, day of year (0 to 365), and a flag that is nonzero if the daylight saving time conversion should be applied.

Return Value

localtime returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

Example

```
#include <time.h>
#include <stdio.h>
int main(void)
{
   time_t timer;
   struct tm *tblock;
   /* gets time of day */
```

```
   timer = time(NULL);
   /* converts date/time to a structure */
   tblock = localtime(&timer);
   printf("Local time is: %s", asctime(tblock));
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.7 mktime

Header File

time.h

Category

Time and Date Routines

Prototype

```
time_t mktime(struct tm *t);
```

Description

Converts time to calendar format.

Converts the time in the structure pointed to by t into a calendar time with the same format used by the time function. The original values of the fields tm_sec, tm_min, tm_hour, tm_mday, and tm_mon are not restricted to the ranges described in the tm structure. If the fields are not in their proper ranges, they are adjusted. Values for fields tm_wday and tm_yday are computed after the other fields have been adjusted.

The allowable range of calendar times is Jan 1 1970 00:00:00 to Jan 19 2038 03:14:07.

Return Value

On success, mktime returns calendar time as described above.

On error (if the calendar time cannot be represented), mktime returns -1.

Example

```
#include <stdio.h>
#include <time.h>
 char *wday[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday", "Unknown"};
int main(void)
{
    struct tm time_check;
    int year, month, day;
/*  Input a year, month and day to find the weekday for */
    printf("Year:  ");
    scanf("%d", &year);
    printf("Month: ");
    scanf("%d", &month);
    printf("Day:   ");
    scanf("%d", &day);
/*  load the time_check structure with the data */
    time_check.tm_year = year - 1900;
    time_check.tm_mon  = month - 1;
```

```
    time_check.tm_mday = day;
    time_check.tm_hour = 0;
    time_check.tm_min  = 0;
    time_check.tm_sec  = 1;
    time_check.tm_isdst = -1;
/*  call mktime to fill in the weekday field of the structure */
    if (mktime(&time_check) == -1)
       time_check.tm_wday = 7;
/*  print out the day of the week */
    printf("That day is a %s\n", wday[time_check.tm_wday]);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.8 stime

Header File

time.h

Category

Time and Date Routines

Prototype

```
int stime(time_t *tp);
```

Description

Sets system date and time.

stime sets the system time and date. tp points to the value of the time as measured in seconds from 00:00:00 GMT, January 1, 1970.

Return Value

stime returns a value of 0.

Example

```
#include <stdio.h>
#include <time.h>
int main(void)
{
  time_t t;
  t = time(NULL);
  printf("Current date is %s", ctime(&t));
  t -= 24L*60L*60L;  /* Back up to same time previous day */
  stime(&t);
  printf("\nNew date is %s", ctime(&t));
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

1.1.2.34.9 _strdate, _wstrdate

Header File

time.h

Category

Conversion Routines, Time and Date Routines

Prototype

```
char *_strdate(char *buf);
```

```
wchar_t *_wstrdate(wchar_t *buf);
```

Description

Converts current date to string.

_strdate converts the current date to a string, storing the string in the buffer buf. The buffer must be at least 9 characters long.

The string has the form MM/DD/YY where MM, DD, and YY are all two-digit numbers representing the month, day, and year. The string is terminated by a null character.

Return Value

_strdate returns buf, the address of the date string.

Example

```
#include <time.h>
#include <stdio.h>
void main(void)
{
  char datebuf[9];
  char timebuf[9];
  _strdate(datebuf);
  _strtime(timebuf);
  printf("Date: %s  Time: %s\n",datebuf,timebuf);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strdate		+		
_wstrdate		+		

1.1.2.34.10 strftime, wcsftime

Header File

time.h

Category

Time and Date Routines

Prototype

```
size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *t);
```

```
size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t *fmt, const struct tm *t);
```

Description

Formats time for output.

strftime formats the time in the argument t into the array pointed to by the argument s according to the fmt specifications. All ordinary characters are copied unchanged. No more than maxsize characters are placed in s.

The time is formatted according to the current locale's LC_TIME category.

Return Value

On success, strftime returns the number of characters placed into s.

On error (if the number of characters required is greater than maxsize), strftime returns 0.

More about strftime

Example

```
#include <stdio.h>
#include <time.h>
#include <dos.h>
int main(void)
{
   struct tm *time_now;
   time_t secs_now;
   char str[80];
   tzset();
   time(&secs_now);
   time_now = localtime(&secs_now);
   strftime(str, 80,
            "It is %M minutes after %I o'clock (%Z)  %A, %B %d 19%y",
            time_now);
   printf("%s\n",str);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strftime	+	+	+	+
wcsftime		+	+	+

1.1.2.34.11 _strtime, _wstrtime

Header File

time.h

Category

Time and Date Routines

Prototype

```
char *_strtime(char *buf);
```

```
wchar_t *_wstrtime(wchar_t *buf);
```

Description

Converts current time to string.

_strtime converts the current time to a string, storing the string in the buffer buf. The buffer must be at least 9 characters long.

The string has the following form:

`HH:MM:SS`

where HH, MM, and SS are all two-digit numbers representing the hour, minute, and second, respectively. The string is terminated by a null character.

Return Value

_strtime returns buf, the address of the time string.

Example

```
#include <time.h>
#include <stdio.h>
void main(void)
{
  char datebuf[9];
  char timebuf[9];
  _strdate(datebuf);
  _strtime(timebuf);
  printf("Date: %s  Time: %s\n",datebuf,timebuf);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_strtime		+		
_wstrtime		+		

1.1.2.34.12 time

Header File

time.h

Category

Time and Date Routines

Prototype

```
time_t time(time_t *timer);
```

Description

Gets time of day.

time gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by timer, provided that timer is not a NULL pointer.

Return Value

time returns the elapsed time in seconds.

Example

```
#include <time.h>
#include <stdio.h>
int main(void)
{
   time_t t;
```

```
    t = time(NULL);
    printf("The number of seconds since January 1, 1970 is %ld",t);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.13 _tzset, _wtzset

Header File

time.h

Category

Time and Date Routines

Prototype

```
void _tzset(void)
```

```
void _wtzset(void)
```

Description

Sets value of global variables _daylight, _timezone, and _tzname.

_tzset is available on XENIX systems.

_tzset sets the _daylight, _timezone, and _tzname global variables based on the environment variable TZ. _wtzset sets the _daylight, _timezone, and _wtzname global variables. The library functions ftime and localtime use these global variables to adjust Greenwich Mean Time (GMT) to the local time zone. The format of the TZ environment string is:

```
TZ = zzz[+/-]d[d][lll]
```

where zzz is a three-character string representing the name of the current time zone. All three characters are required. For example, the string “PST” could be used to represent Pacific standard time.

[+/-]d[d] is a required field containing an optionally signed number with 1 or more digits. This number is the local time zone’s difference from GMT in hours. Positive numbers adjust westward from GMT. Negative numbers adjust eastward from GMT. For example, the number 5 = EST, +8 = PST, and -1 = continental Europe. This number is used in the calculation of the global variable _timezone. _timezone is the difference in seconds between GMT and the local time zone.

lll is an optional three-character field that represents the local time zone, daylight saving time. For example, the string “PDT” could be used to represent pacific daylight saving time. If this field is present, it causes the global variable _daylight to be set nonzero. If this field is absent, _daylight is set to zero.

If the TZ environment string isn’t present or isn’t in the preceding form, a default TZ = “EST5EDT” is presumed for the purposes of assigning values to the global variables _daylight, _timezone, and _tzname. On a Win32 system, none of these global variables are set if TZ is null.

The global variables _tzname[0] and _wtzname[1] point to a three-character string with the value of the time-zone name from the TZ environment string. _tzname[1] and _wtzname[1] point to a three-character string with the value of the daylight saving time-zone name from the TZ environment string. If no daylight saving name is present, _tzname[1] and _wtzname[1] point to a null string.

Return Value

None.

Example

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   time_t td;
   putenv("TZ=PST8PDT");
   tzset();
   time(&td);
   printf("Current time = %s\n", asctime(localtime(&td)));
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_tzset	+	+		
_wtzset		+		

1.1.2.34.14 _utime, _wutime

Header File

utime.h

Category

Input/output Routines

Prototype

```
int _utime(char *path, struct utimbuf *times);
```

```
int _wutime(wchar_t *path, struct _utimbuf *times);
```

Description

Sets file time and date.

_utime sets the modification time for the file path. The modification time is contained in the utimbuf structure pointed to by times. This structure is defined in utime.h, and has the following format:

```
struct utimbuf {
```

```
time_t actime; /* access time */
```

```
time_t modtime; /* modification time */
```

```
};
```

The FAT (file allocation table) file system supports only a modification time; therefore, on FAT file systems _utime ignores actime and uses only modtime to set the file's modification time.

If times is NULL, the file's modification time is set to the current time.

_wutime is the Unicode version of _utime. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform identically.

Return Value

On success, _utime returns 0.

On error, it returns -1, and sets the global variable errno to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Example

```
/* Copy timestamp from one file to another */
#include <sys\stat.h>
#include <utime.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  struct stat src_stat;
  struct utimbuf times;
  if(argc != 3) {
    printf( "Usage: copytime <source file> <dest file>\n" );
    return 1;
  }

  if (stat(argv[1],&src_stat) != 0) {
    perror("Unable to get status of source file");
    return 1;
  }

  times.modtime = times.actime = src_stat.st_mtime;
  if (utime(argv[2],&times) != 0) {
    perror("Unable to set time of destination file");
    return 1;
  }
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_utime	+	+		
_wutime		+		

1.1.2.34.15 clock_t

Header File

time.h

Syntax

```
typedef long clock_t;
```

Description

Defines the data type returned by the clock function.

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

1.1.2.34.16 _daylight

Header File

time.h

Syntax

```
extern int _daylight;
```

Description

_daylight is used by the time and date functions. It is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. _daylight is initialized from the values specified in the TZ environment variable and is set to 1 if the daylight savings time conversion should be applied. If TZ is not set, the value of _daylight is obtained from the operating system.

1.1.2.34.17 _timezone

Header File

time.h

Syntax

```
extern long _timezone;
```

Description

_timezone is used by the time-and-date functions. It is calculated by the tzset function; it is assigned a long value that is the difference, in seconds, between the current local time and Greenwich mean time.

On Win32, the value of _timezone is obtained from the operating system.

1.1.2.34.18 tm

Header File

time.h

Syntax

```
struct tm {
int tm_sec; /* Seconds */
int tm_min; /* Minutes */
int tm_hour; /* Hour (0--23) */
int tm_mday; /* Day of month (1--31) */
int tm_mon; /* Month (0--11) */
int tm_year; /* Year (calendar year minus 1900) */
```

```
int tm_wday; /* Weekday (0--6; Sunday = 0) */
int tm_yday; /* Day of year (0--365) */
int tm_isdst; /* 0 if daylight savings time is not in effect) */
};
```

Description

A structure defining the time, broken down into increments.

tm is used by the functions asctime, gmtime, localtime, mktime, and strftime.

Example

1.1.2.34.19 _tzname,_wtzname

Header File

time.h

Syntax

```
extern char * _tzname[2]
extern wchar_t *const _wtzname[2]
```

Description

The global variable _tzname is an array of pointers to strings containing abbreviations for time zone names. _tzname[0] points to a three-character string with the value of the time zone name from the TZ environment string. The global variable _tzname[1] points to a three-character string with the value of the daylight saving time zone name from the TZ environment string. If no daylight saving name is present, _tzname[1] points to a null string.

On Win32, the value of _tzname is obtained from the operating system.

1.1.2.35 typeinfo.h

The following functions, macros, and classes are provided in `typeinfo.h`:

1.1.2.35.1 bad_cast class

Header File

typeinfo.h

Description

When dynamic_cast fails to make a cast to reference, the expression can throw bad_cast. Note that when dynamic_cast fails to make a cast to pointer type, the result is the null pointer.

1.1.2.35.2 bad_typeid class

Header File

typeinfo.h

Description

When the operand of typeid is a dereferenced null pointer, the **typeid** operator can throw bad_typeid.

1.1.2.35.3 type_info class

Header File

typeinfo.h

Description

Provides information about a type.

Constructor

Only a private constructor is provided. You cannot create type_info objects. By declaring your objects to be _ _rtti types, or by using the -RT compiler switch, the compiler provides your objects with the elements of type_info. type_info references are generated by the typeid operator.

Public Member Functions

Operators

1.1.2.36 utime.h

The following functions, macros, and classes are provided in `utime.h`:

1.1.2.37 values.h

The following functions, macros, and classes are provided in `values.h`:

1.1.2.37.1 MAXxxxx #defines (integer data types)

Header File

values.h

Description

Maximum values for integer data types

Name	Meaning
MAXSHORT	Largest short
MAXINT	Largest int
MAXLONG	Largest long

1.1.2.37.2 BITSPERBYTE #define

Header File

values.h

Description

Number of bits in a byte.

1.1.2.37.3 Float and Double Limits

Header File

values.h

Description

UNIX System V compatible:

_LENBASE	Base to which exponent applies

Limits for double float values

_DEXPLEN	Number of bits in exponent
DMAXEXP	Maximum exponent allowed
DMAXPOWTWO	Largest power of two allowed
DMINEXP	Minimum exponent allowed
DSIGNIF	Number of significant bits
MAXDOUBLE	Largest magnitude double value
MINDOUBLE	Smallest magnitude double value

Limits for float values

_FEXPLEN	Number of bits in exponent
FMAXEXP	Maximum exponent allowed
FMAXPOWTWO	Largest power of two allowed
FMINEXP	Minimum exponent allowed
FSIGNIF	Number of significant bits
MAXFLOAT	Largest magnitude float value
MINFLOAT	Smallest magnitude float value

1.1.2.37.4 HIBITx #defines

Header File

values.h

Description

Bit mask for the high (sign) bit of standard integer types.

Name	Meaning
HIBITS	For type short
HIBITI	For type int
HIBITL	For type long

1.1.3 C++ Compiler Errors And Warnings

This topic explains the error and warning messages emitted by bcc32.

1.1.3.1 List Of All C++ Compiler Errors And Warnings

This section describes the RAD Studio C++ compiler error and warning messages.

1.1.3.1.1 Cannot take address of member function 'function' (E2196)

An expression takes the address of a class member function, but this member function was not found in the program being debugged. The evaluator issues this message.

1.1.3.1.2 Explicit specialization of 'specifier' is ambiguous: must specify template arguments (E2506)

In the following code, explicit template arguments are necessary:

```
template<class T> void foo(T);
template<class T> void foo(T *);
template<> void foo(int *); // error, must say 'foo<int>' or 'foo<int *>'
```

1.1.3.1.3 Array dimension 'specifier' could not be determined (E2483)

If, during instantiation of a type, an array dimension cannot be computed—usually this is due to some other error which would be reported—then this error will result.

For example, if an array dimension is dependent upon a template parameter but an error occurs while it is being parsed and the template argument being substituted does not yield a legal constant expression, this error is displayed.

1.1.3.1.4 Value out of range (E2509)

The inline assembler has detected a numeric overflow in one of your expressions. Make sure all of your numbers can fit in 32 bits.

1.1.3.1.5 Operand size mismatch (E2510)

Help is not available for this item.

1.1.3.1.6 __declspec(delphireturn) class 'class' must have exactly one data member (E2050)

This is an internal compiler error. A class marked as a delphireturn class has more than one non-static data member.

1.1.3.1.7 Unrecognized option, or no help available (E2530)

You have entered a command element that the C++ compiler cannot parse, or the option you entered has no associated help. Try again.

1.1.3.1.8 Option 'name' cannot be set via 'name' (E2527)

An attempt was made in a source file to set an option that cannot be set after either parsing or compiling of the file starts. Instead, set this option on the command line or in a `.cfg` file.

For example, if a source file contains a `#pragma option push –v`, you need to remove the push or set /unset this option either on the command line or in a `.cfg` file.

1.1.3.1.9 Value after -g or -j should be between 0 and 255 inclusive (E2074)

Both the -g and the -j command line options can be followed by an optional number. The compiler expects this number to be between 0 and 255 inclusive.

1.1.3.1.10 Option 'name' must be set before compilation begins (E2528)

An attempt was made in a source file to set an option that must be set before compiling starts. Instead, set this option on the command line, in a `.cfg` file, or at the top of the source file before the line `int foo();` .

1.1.3.1.11 Properties may only be assigned using a simple statement, e.g. \"prop = value;\" (E2492)

Assignments to properties should be made in simple assignment statements. If property assignments could become Lvalues, which happens when property assignments are embedded in larger statements, the getter is called to create the Lvalue, with all the side effects that getter causes. The compiler allows only one call to either the getter or the setter in a statement.

For example:

```
class myClass
{
  int X;
  public:
  int __property x = { read=getx, write=putx };
  int getx() { return X; }
  void putx(int val) { X = val; }
} OneClass;
int value(int);
int main()
{
  return value(OneClass.x = 4);  // This causes an error
}
```

1.1.3.1.12 Invalid template declarator list (E2100)

It is illegal for a declarator list to follow a template class declaration. For example:

```
template<class T>
class foo {
} object_name;  // This causes an error
```

1.1.3.1.13 Cannot use template 'template' without specifying specialization parameters (E2102)

The generic form of a template must be referenced using specialization parameters. For example, for a template class named foo, taking two template parameters, then a legal reference might have the form

```
foo<int, char>
```

Referring to the template as just foo is legal in only two circumstances:

- When passing the template name as a template template argument
- While declaring the members of that class template, to refer to the enclosing template type

For example:

```
template<class T>
class foo
{
public:
    foo();       // legal use of bare template name
    foo& operator=(const foo&);
};
foo<foo> x;   // error: not a template template argument
foo y;    // error: needs specialization parameters
```

1.1.3.1.14 Invalid use of template 'template' (E2107)

This error results when attempting to use a template template parameter in any way other than to reference a template specialization, or to pass that parameter in turn as a template template argument to another template. For example:

```
template<template<class T> class U>
class foo;
template<template<class T> class U>
class bar
{
U x;// error: not a specialization
U<U> y;// ok: used as a specialization, and as a
// template template argument
U<bar> z;// ok: used to reference a specialization
};
```

1.1.3.1.15 'template' qualifier must specify a member template name (E2105)

When parsing code that depends in some way upon a template parameter, it is sometimes impossible to know whether a member name will resolve to a template function name, or a regular parameter. In the following code, a 'template' qualifier is required in order to know if the '<' (less-then) operator should be parsed as the beginning character of a template argument list, or as a regular less-than operator:

```
template<class T>
void foo(T a)
{
a.member<10>();
}
```

Although it may be apparent to the reader what is meant, the compiler does not know that "member" refers to a member template function, and it will parse the line of code as follows:

```
a.member < (10>());
```

In order to tell the compiler that the less-than character begins a template argument list, the 'template' qualifier is needed:

```
a.template member<10>();   // "member" must be a member template
```

If the 'template' qualifier is used in a situation where "member" does not resolve to a member template, the above error will result.

1.1.3.1.16 Explicit instantiation requires an elaborated type specifier (i.e.,"class foo<int>") (E2505)

The following code is illegal:

```
template<class T> class foo;
template foo<int>;            // missing `class' keyword
```

See Also

Implicit And Explicit Template Functions (see page 144)

1.1.3.1.17 Information not available (E2066)

Help is not available for this item.

1.1.3.1.18 pragma checkoption failed: options are not as expected (E2471)

You can use #pragma checkoption to check that certain switches are in the state that you expect. If #pragma checkoption detects that a switch is not in the expected state, the compiler displays this error.

You can use the following syntax:

```
#pragma checkoption <options>
```

For example:

```
#pragma checkoption -O2
```

OR

```
#pragma checkoption -C -O2
```

The compiler will check if the option(s) are turned on. If all are turned on, nothing happens. If at least one is not turned on, this error is displayed.

1.1.3.1.19 'dynamic' can only be used with non-template member functions (E2504)

You tried to use dynamic with a template member function. Dynamic functions are allowed for classes derived from TObject. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of its ancestors are searched until the function is found.

1.1.3.1.20 '__far16' may only be used with '__pascal' or '__cdecl' (E2191)

This is an internal compiler error. The compiler emits this message if the keyword __far16 is mixed with one of the keywords __pascal or __cdecl, all in the same declaration.

1.1.3.1.21 Template friend function 'function' must be previously declared (E2199)

Not used

1.1.3.1.22 Error resolving #import: problem (E2502)

Where `problem` can be any of the following relating to problems with the various attributes of #include (see page 152):

unexpected import directive value attribute 'attribute'A value was supplied for the indicated attribute. None was expected.

missing ')' in import directive attribute 'attribute'The value for the indicated attribute was incorrectly specified : a closing parenthesis is missing.

unrecognized import directive attribute 'attribute'The indicated token is not a legitimate attribute for the #import directive.

invalid values for raw_property_prefixes attributeThe values for the raw_property_prefixes attribute were incorrectly specified.

unexpected duplicate property 'property'The indicated #import attribute was specified more than once -- this is an error.

unexpected duplicate get method for property 'property'The get method for the indicated property was specified more than once.

unexpected duplicate put method for property 'property'The put method for the indicated property was specified more than once.

unexpected duplicate put-reference method for property 'property'The put-reference method for the indicated property was specified more than once.

no return value specified for property get method 'method'The indicated property get method does not supply the correct return type.

no return value specified for property put method 'method'The indicated property put method does not supply the correct return type.

could not load type library in 'filename'The indicated type library could not be opened.

could not obtain type library nameThe compiler could not obtain obtain a library name for the current type library

See Also

#include (see page 152)

1.1.3.1.23 Unable to open import file 'filename' (E2501)

This error occurs when you use:

```
#import "somefile.h"
```

and the file you are trying to import doesn't exist or can't be found by the compiler.

1.1.3.1.24 Invalid MOM inheritance (E2066)

The compiler issues this error if the currently compiled class doesn't have the same MOM (Microsoft Object Model) related flags set as its direct parent.

This compiler error message is deprecated.

1.1.3.1.25 Unrecognized __declspec modifier (E2494)

A _declspec modifier was given that is not valid.

1.1.3.1.26 Invalid GUID string (E2493)

The GUID string does not have the form of a Globally Unique Identifier.

1.1.3.1.27 Invalid __declspec(uuid(GuidString)) format (E2499)

This error happens when you used the wrong format to define your GuidString. GUIDs are defined for structs in the following way:

```
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo{};
```

You would get the above mentioned error for instance from:

```
class __declspec(uuid(19a76fe0-7494-11d0-8816-00a0c903b83c)) foo{}; //Missing quotes
```

or

```
class __declspec(uuid"7494-11d0-8816-00a0c903b83c")) foo{}; // Missing left parentheses
```

1.1.3.1.28 Invalid call to uuidof(struct type|variable) (E2496)

The uuidof operator was given an incorrect argument.

1.1.3.1.29 Unterminated macro argument (E2511)

A macro argument that was started on the line listed has not been properly terminated

1.1.3.1.30 Maximum option context replay depth exceeded; check for recursion (E2489)

If this error is triggered, it means that recursive template instantiation has gone too deep. Check for compile-time recursion in your program, and limit it to no more than 256 levels.

1.1.3.1.31 Maximum token reply depth exceeded; check for recursion (E2488)

If this error is triggered, it means that recursive template instantiation has gone too deep. Check for compile-time recursion in your program, and limit it to no more than 256 levels.

1.1.3.1.32 Maximum VIRDEF count exceeded; check for recursion (E2491)

Too many VIRDEF symbols were allocated. The compiler imposes a limit to the number of VIRDEFs allowed per translation unit. Currently this limit is in the order of 16384 VIRDEFs.

One way this could happen is if a program has more than 16384 functions.

1.1.3.1.33 In-line data member initialization requires an integral constant expression (E2230)

Static const class members, which are initialized in the body of the class, have to be initialized with a constant expression of integral type.

1.1.3.1.34 VCL style classes need virtual destructors (E2241)

Destructors defined in VCL style classes have to be virtual.

Example

```
struct__declspec(delphiclass) vclclass1
{
~vclclass1() {}           // Error
};
struct__declspec(delphiclass) vclclass2
{
virtual ~vclclass2() {}   // OK
};
```

1.1.3.1.35 Anonymous structs/unions not allowed to have anonymous members in C++ (E2524)

The C++ compiler requires that the members of an anonymous struct or union be named.

1.1.3.1.36 x is not abstract public single inheritance class hierarchy with no data (E2246)

Internal compiler error. In some cases, the compiler will enforce restrictions on a class hierarchy. In this case, the restrictions would be that all classes are abstract classes, and all classes only have one parent.

1.1.3.1.37 = expected (E2249)

The compiler expected an equal sign in the position where the error was reported but there was none. This is usually a syntax error or typo.

1.1.3.1.38 First base must be VCL class (E2267)

Internal compiler error. In some cases, the compiler will enforce restrictions on a class hierarchy. In this case, the restrictions would be that the first parent of a class is a Delphi style class.

1.1.3.1.39 Cannot declare a member function via instantiation (E2472)

If a declaration within a template class acquires a function type through a type dependent on a template-parameter and this results in a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. For example:

```
template<class T>
struct A {
static T t;
```

```
};
typedef int function();
A<function> a;// error: would declare A<function>::t
// as a static member function
```

Another example:

In the example below, the template member 'a' has type 'T'. If the template is instantiated with T as a function type, it implies that 'a' is therefore a member function. This is not allowed and the error message is displayed.

```
template<T& x> class foo { T a; }
int func(int);
template class foo<func>;
```

1.1.3.1.40 Cannot explicitly specialize a member of a generic template class (E2515)

You are trying to make a generic template into a specialized member. For example, the following code is illegal:

```
    template<typename T>
class foo {
template<typename U>
      class bar {
      };
    };
template<typename T>
template<>
class foo<T>::bar<char> {
};
```

The second declaration in the example is an error, because it tries to explicitly specialize bar<char> within foo<T>.

1.1.3.1.41 'function' cannot be declared as static or inline (E2474)

You attempted to declare a symbol as static or inline and this type of symbol cannot be defined as static or inline. Certain functions, like 'main' and 'WinMain' cannot be declared static or inline. 'main' is the entrypoint of console applications, and 'WinMain' is the entry point of Windows applications.

For example, this error is displayed in the following cases:

```
static int main()  // This causes an error
{}
```

or

```
inline int main() { return 0; }
```

1.1.3.1.42 Non-const function 'function' called for const object (E2522)

Data type mismatch resulting in an erroneous function call. The object of the call (a non-const function) should be a const object.

1.1.3.1.43 Non-volatile function 'name' called for volatile object (E2523)

Data type mismatch. The error is the result of an erroneous function call. The object of the call (a non-volatile function) should be a volatile object.

1.1.3.1.44 Cannot emit RTTI for 'parameter' in 'function' (E2513)

The compiler issues this error if it cannot generate RTTI information for the return type of a parameter. See Runtime type

information for more information.

1.1.3.1.45 Cannot emit RTTI for return type of 'function' (E2512)

The compiler issues this error if the it cannot generate RTTI information for the return type of a function. See Runtime type information for more information.

1.1.3.1.46 Need previously defined struct GUID (E2498)

This happens when you use the __uuidof operator without including a header that defines the GUID struct. So the following program code would display this error:

```
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo{};
int main()
{
    __uuidof(foo);
    return 0;
}
```

And you would fix it as follows:

```
#include <windows.h>    // Will pull in struct GUID
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo{};
int main()
{
    __uuidof(foo);
    return 0;
}
```

1.1.3.1.47 'class' is not a direct base class of 'class' (E2507)

The first type is not a direct base class of the second type. A direct base class refers to the immediate derivations of that class, and not the derivations of its subclasses.

1.1.3.1.48 Too many candidate template specializations from 'specifier' (E2295)

When reference a class template specialization, it is possible that more than one possible candidate might result from a single reference. This can only really happen among class partial specializations, when more than one partial specialization is contending for a possible match:

```
template<class T, class U>
class foo;
template<class T>
class foo<T, T *>;
template<class T>
class foo<T *, T>;
foo<int *, int *> x;  // error: which partial specialization to use?
```

In this example, both partial specializations are equally valid, and neither is more specialized than the other, so an error would result.

1.1.3.1.49 'function' cannot be a template function (E2475)

Certain functions, like 'main' and 'WinMain' cannot be declared as a template function. 'main' is the entrypoint of console applications, and 'WinMain' is the entry point of Windows applications.

For example:

```
template <class T> int main()  // This causes an error
{}
```

See Also

functiontemplates.xml (see page 143)

1.1.3.1.50 Cannot generate template specialization from 'specifier' (E2299)

This error will result if an attempt is made to reference a template class or function in a manner which yields no possible candidate specializations. For example:

```
template<class T>
class foo;
foo<10> x;       // error: arguments aren't valid for 'foo'
```

1.1.3.1.51 Could not generate a specialization matching type for 'specifier' (E2300)

This error is no longer generated by the compiler.

1.1.3.1.52 No GUID associated with type:'type' (E2497)

A variable or type was used in a context requiring a GUID, but the type does not have a GUID associated with it. GUIDs are associated with types using _declspec (uuid(GUID)).

1.1.3.1.53 Path 'path' exceeds maximum size of 'n' (E2529)

In looking up include files, the C++ compiler has encountered a file whose path name contains more characters than are allowed in the Windows maximum. Rename the path to a shorter name.

1.1.3.1.54 You must define _PCH_STATIC_CONST before including xstring to use this feature (E2525)

You attempted to use a feature defined in xstring, part of the Dinkumware standard C++ library. The C++ compiler could not generate a precompiled header because there is a constant (defined in xstring) in the header. If you want to include xstring, you should first set the define _PCH_STATIC_CONST.

1.1.3.1.55 Property 'name' uses another property as getter/setter; Not allowed (E2526)

Properties typically have both a getter and a setter, but a property cannot serve as either the getter or setter of another property.

1.1.3.1.56 Published property access functions must use __fastcall calling convention (E2008)

The calling convention for access functions of a property (read, write, and stored) declared in a __published section must be __fastcall. This also applies to hoisted properties.

Example

```
struct__declspec(delphiclass) clxclass{int__fastcall Getter1(void); int__cdecl
Getter2(void); __published:int __property ip1 = {read = Getter1};    // OKint __property ip2 =
{read = Getter2};    // Error};
```

1.1.3.1.57 Function call terminated by unhandled exception 'value' at address 'addr' (E2122)

This message is emitted when an expression you are evaluating while debugging includes a function call that terminates with an unhandled exception. For example, if in the debugger's evaluate dialog, you request an evaluation of the expression foo()+1 and the execution of the function foo() causes a GP fault, this evaluation produces the above error message.

You may also see this message in the watches window because it also displays the results of evaluating an expression.

1.1.3.1.58 Redefinition of uuid is not identical (E2495)

GUID's attached to structs have to be the same across multiple declarations and definitions of the same struct. So the following example would cause this error:

```
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo;
class __declspec(uuid("00000000-7494-11d0-8816-00a0c903b83c")) foo{}
```

1.1.3.1.59 __declspec(selectany) is only for initialized and externally visible variables (E2500)

You cannot use __declspec(selectany) with static variables, unitialized variables, etc.

1.1.3.1.60 String constant expected (E2482)

The compiler expected a string constant at this location but did not receive one.

This error is no longer generated by the compiler.

1.1.3.1.61 Unexpected string constant (E2481)

There are times when the compiler does not expect a string constant to appear in the source input. For example:

```
class foo { "Hello"; };
```

1.1.3.1.62 Cannot involve parameter 'parameter' in a complex partial specialization expression (E2386)

When declaring or defining a template class partial specialization, it is illegal to involve any of the non-type template parameters in complex expressions. They may only be referenced by name. For example:

```
template<class T, int U>
class foo;
template<int U>
class foo<char, U * 3>;// error: "U * 3" is a complex expression
template<int U>
class foo<char, U>;// OK: "U" is a simple, by-name expression
```

1.1.3.1.63 Partial specializations may not specialize dependent non-type parameters ('parameter') (E2387)

A partial specialization may not use a template parameter in its specialization argument list which is dependent on another type parameter. For example:

```
template<class T, int U>
class foo;
template<class T, T U>
class foo<T *, U>;      // error: 'U' is type-dependent
```

1.1.3.1.64 Argument list of specialization cannot be identical to the parameter list of primary template (E2388)

When declaring a partial specialization, its specialization argument list must differ in some way from its basic parameter list. For example:

template<class T>

```
class foo;
template<class T>
class foo<T *>;     // OK: is more specialized than primary template
template<class T>
class foo<T>;       // error: identical to primary template
```

1.1.3.1.65 Mismatch in kind of substitution argument and template parameter 'parameter' (E2389)

When referencing a template specialization, all type parameters must be satisfied using type arguments, all non-type parameters require non-type arguments, and all template template parameters require either a template name, or another template template argument. Mismatching these requirements in any way will trigger the above error. For example:

```
template<class T, int U, template<class V> class W>
class foo;
foo<char, 10, foo> x;    // OK: all parameter kinds match
foo<10, char, int> y;    // error: no parameter kinds match at all!
```

1.1.3.1.66 Cannot involve template parameters in complex partial specialization arguments (E2480)

A partial specialization cannot reference other template parameters in a nonvalue argument expression, unless it is simply a direct reference to the template parameter. For example:

```
template<int A, int B, int C> class foo;
template<int A> class foo<A+5, A, A+10>;
```

The partial specialization has two illegal arguments. 'A+5' is a complex expression because it uses 'A' in a manner other than as merely a direct argument. The reference to plain 'A' in the second argument is fine, but the third argument is also illegal because it references 'A' in a complex manner as well.

1.1.3.1.67 Template instance 'template' is already instantiated (E2392)

There are two ways to trigger this error. If –A is enabled (ANSI compliant mode), then attempting to explicitly instantiate a template specialization which has already been instantiated (either implicitly or explicitly) will cause this error. Regardless of –A, attempting to explicitly specialize a template specialization which has already been either implicit or explicitly instantiated will

always trigger this error. For example:

```
template<class T>
class foo;
foo<char> x;     // causes implicit instantiation of "foo<char>"
template<>
class foo<char> { };   // error: "foo<char>" already instantiated
template class foo<char>;   // error in –A mode, otherwise a warning
```

1.1.3.1.68 Cannot take the address of non-type, non-reference template parameter 'parameter' (E2393)

A template parameter has no address, and is not associated with any real "object". Therefore, to take its address, or attempt to assign to it, has no meaning. For example:

```
template<int U>
void foo()
{
int *x = &U;// error: cannot take address of parameter
}
```

1.1.3.1.69 Cannot reference template argument 'arg' in template class 'class' this way (E2399)

The compiler no longer generates this error.

1.1.3.1.70 Template argument cannot have static or local linkage (E2397)

Only integral constant expressions, and the address of global variables with external linkage, may be used as template arguments. For example:

```
template<char *x>
class foo;
const char *p = "Hello";
extern char *q;
foo<p> x;// OK: "p" is visible to the outside
foo<q> y;// OK: "q" is also globally visible
foo<"Hello"> z;// error: string literal has static linkage
```

1.1.3.1.71 Cannot use address of array element as non-type template argument (E2485)

Non-type template arguments may only be of integral type, or the address of a global variable. They cannot be the address of an array element. For example:

```
int x[100];
template<int T>
class foo;
foo<&x[0]> y;// error: not an integral or global address
```

1.1.3.1.72 Illegal base class type: formal type 'type' resolves to 'type' (E2402)

When instantiating a template class definition, if it is found that a declared base class does not resolve to an accessible class type, this error will result. For example:

```
template<class T>
class foo : public T { };
foo<int> x;// error: "int" is not a valid base class
```

```
foo<bar> y;// error: "bar" is an unknown type
```

1.1.3.1.73 Dependent call specifier yields non-function 'name' (E2403)

The compiler no longer generates this error.

1.1.3.1.74 Dependent type qualifier 'qualifier' has no member type named 'name' (E2404)

If a template declaration references a member of a dependent type, it is only possible to alert the user to the non-existence of this member during type instantiation for a given set of template arguments. For example:

```
template<class T>
class foo
{
typename T::A x;     // we expect that "A" is a member type
};
struct bar { };
foo<bar> y;// error: "bar" has no member type named "A"
```

1.1.3.1.75 Dependent template reference 'identifier' yields non-template symbol (E2405)

If a template specialization reference within a template declaration yields a reference to a non-template during type instantiation, the above error will result. For example:

```
template<class T>
class foo
{
typename T::template A<int> x;  // "A" must be a member template
};
struct bar {
struct A { };
};
foo<bar> y;// error: bar::A is a non-template class!
```

1.1.3.1.76 Dependent type qualifier 'qualifier' is not a class or struct type (E2406)

If a dependent name reference within a template declaration results in a non-struct member qualification at instantiation time, the above error will result. For example:

```
template<class T>
class foo
{
typename T::A x;     // we expect that "A" is a member type
};
foo<int> y;// error: "int" cannot be qualified; not a class
```

1.1.3.1.77 Dependent type qualifier 'qualifier' has no member symbol named 'name' (E2407)

If a template declaration references a member of a dependent type, it is only possible to alert the user to the non-existence of this member during type instantiation for a given set of template arguments. For example:

```
template<class T>
class foo
```

```
{
foo(int *a = T::A);     // we expect that "A" is a data member
};
struct bar { };
foo<bar> y;// error: "bar" has no member named "A"
```

1.1.3.1.78 Default values may be specified only in primary class template declarations (E2408)

Template functions, and class partial specializations, may not use default expressions in their template parameter lists. Only primary template declarations may do this. For example:

```
template<class T = int>
class foo;// OK: primary class template
template<class T = int>
void bar();// error: template function
template<class T = int>
class foo<T *>;// error: partial specialization
```

1.1.3.1.79 Cannot find a valid specialization for 'specifier' (E2409)

This error is no longer generated by the compiler.

1.1.3.1.80 Cannot use address of class member as non-type template argument (E2486)

Non-type template arguments may only be of integral type, or the address of a global variable. They cannot be the address of a class member. For example:

```
struct bar {
int x;
} y;
template<int T>
class foo;
foo<&y.x> z;// error: not an integral or global address
```

1.1.3.1.81 Missing template parameters for friend template 'template' (E2410)

If a friend template is declared, but no template parameters are specified, this error will result. For example:

```
template<class T>
class foo;
class bar {
friend class foo;// error: forgot template parameters!
};
```

1.1.3.1.82 Destructors cannot be declared as template functions (E2414)

Destructors cannot be templates. For example:

```
class foo {
template<class T>
virtual ~foo();// error: don't try this at home!
};
```

1.1.3.1.83 Declaration of member function default parameters after a specialization has already been expanded (E2411)

If a member function of a class template is declared, and then a specialization of that class implicitly instantiated, and later that member function defined with default parameters specified, the above error will result. For example:

```
template<int i>
class foo {
void method(int a, int b = i);
};
foo<10> x;
template<int i>
void foo<i>::method(int a = i, int b);  // error!
```

1.1.3.1.84 Attempting to bind a member reference to a dependent type (E2412)

The compiler no longer generates this error.

1.1.3.1.85 Invalid explicit specialization of 'specifier' (E2473)

Attempting to explicitly specialize a static data member or any non-template will cause this error.

1.1.3.1.86 Specialization within template classes not yet implemented (E2490)

Explicit and partial specialization of member template classes and functions within template classes and nested template classes, is not supported.

1.1.3.1.87 Invalid template function declaration (E2416)

The compiler no longer generates this error.

1.1.3.1.88 Cannot specify template parameters in explicit specialization of 'specifier' (E2417)

The compiler no longer generates this error.

1.1.3.1.89 Maximum instantiation depth exceeded; check for recursion (E2418)

The compiler only supports 256 levels of instantiation before it will trigger this error. The main problem is in controlling stack depth, because the parser uses recursive functions to manage type instantiation. Here is an example that would produce such an error:

```
template<int T>
class foo {
public:
static const int x = foo<T – 1>::x;
};
template<int T>
class foo<1> {
public:
static const int x = 1;
};
```

```
int main() {
int y = foo<100000>::x;// error: instantiation depth exceeded
}
```

1.1.3.1.90 Explicit instantiation can only be used at global scope (E2420)

Explicit instantiation cannot be specified at any level other than namespace or global scope. For example:

```
template<class T>
class foo { };
template class foo<char>;// OK: at global scope
int main() {
template class foo<int>:// error: local scope
}
```

1.1.3.1.91 Argument kind mismatch in redeclaration of template parameter 'parameter' (E2422)

If a template is declared at one point in the translation unit, and then redeclared with template parameters of a different kind at another location, this error will result. For example:

```
template<class T>
class foo;
// ? time passes ?
template<int T>
class foo;// error: type vs. non-type parameter
```

1.1.3.1.92 Cannot have both a template class and function named 'name' (E2479)

No other function or type may have the same name as a template class. For example:

```
void foo();// error: there is a template class named "foo"
template<class T>
class foo;
```

1.1.3.1.93 The name of template class 'class' cannot be overloaded (E2484)

Attempting to declare a function that overrides the name of a template class will cause this error. For example:

```
template<class T>
class foo;
void foo();// error: there is a template class named "foo"
```

1.1.3.1.94 Explicit specialization of 'specifier' requires 'template<>' declaration (E2426)

According to the standard, explicit specialization of any template now always require the "template<>" declarator syntax. For example:

```
template<class T>
class foo;
template<>
class foo<char>;// OK: "template<>" was provided
class foo<int>;// error: "template<>" required
```

1.1.3.1.95 Cannot specify default function arguments for explicit specializations (E2487)

An explicit specialization of a function may not declare default function arguments. For example:

```
template<class T>
void foo(T a);
template<>
void foo<int>(int a = 10);// error: default value not allowed
```

1.1.3.1.96 'main' cannot be a template function (E2427)

'main' cannot be declared as a template function. 'main' is the entry point of a console application, and it should be declared as a regular __cdecl function.

This error message should not occur because it has been replaced with another one (E2475).

1.1.3.1.97 Explicit specialization or instantiation of non-existing template 'template' (E2423)

Attempting to explicit specialize or instantiate a template which does not exist is clearly illegal. For example:

```
template<class T>
class foo;
template class bar<char>;// error: what is "bar"??
template<>
class bar<int> { };// error: there's that "bar" again?
```

1.1.3.1.98 Number of template parameters does not match in redeclaration of 'specifier' (E2430)

If a template is redeclared with a different number of template parameters, this error will result. For example:

```
template<class T>
class foo;
template<class T, int U>
class foo;// error: parameter count mismatch!
```

1.1.3.1.99 Too few template parameters were declared for template 'template' (E2477)

If a member declaration or definition occurs outside of a template class, and that outer declaration uses a different number of template parameters than the parent class, this error will result. For example:

```
template<class T, class U>
class foo {
void method();
};
template<class T>
void foo<T>::method() { }// error: too few template parameters!
```

1.1.3.1.100 Too many template parameters were declared for template 'template' (E2478)

If a member declaration or definition occurs outside of a template class, and that outer declaration uses a different number of

template parameters than the parent class, this error will result. For example:

```
template<class T, class U>
class foo {
void method();
};
template<class T, class U, class V>
void foo<T, U, V>::method() { }// error: too many parameters!
```

1.1.3.1.101 Non-type template parameters cannot be of floating point, class, or void type (E2431)

Non-type template parameters are restricted as to what type they may be. Floating point, class and void types are illegal. For example:

```
template<float U>
class foo;// error: "U" cannot be of "float" type
```

1.1.3.1.102 Not a valid partial specialization of 'specifier' (E2429)

Internal compiler error.

1.1.3.1.103 Template declaration missing template parameters ('template<...>') (E2434)

In a context where at least one template parameter is clearly required, if none are found this error will result. For example:

```
template<class T, template<> class U>
class foo;// error: template template parameters require
// at least one actual parameter to be declared
```

1.1.3.1.104 Too many template parameter sets were specified (E2435)

If a member template is being defined outside of its parent class, and too many template parameter sets are declared, this error will result. For example:

```
template<class T>
class foo
{
template<class U>
void method(U a);
};
template<class T> template<class U> template<class V>
void foo<T>::method(U a);// error: too many parameter sets!
```

1.1.3.1.105 'typename' should be followed by a qualified, dependent type name (E2437)

Whenever the "typename" keyword is used in a template declaration or definition, it should always name a dependent type. For example:

```
struct bar { };
template<class T>
class foo {
typename T::A *x;// OK: names a qualified type
typename T y;// error: not a qualified type
typename bar z;// error: not a dependent type
};
```

1.1.3.1.106 Template template arguments must name a class (E2438)

A template template parameter must always declare a new class name. For example:

```
template<template<class T> int U>
class foo;// error: "U" is not a class tag name
template<template<class T> class V>
class bar;// OK: "V" is a class tag name
```

1.1.3.1.107 Default type for template template argument 'arg' does not name a primary template class (E2436)

If a template template parameter is to have a default type, that type must either be a generic template class name, or another template template parameter.

```
template<class T>
class foo;
template<template<class T> class U = foo>
class bar;// OK: "foo" is a qualifying primary template
template<template<class T> class U = int>
class baz;// error: "int" is not a template class
```

1.1.3.1.108 'typename' is only allowed in template declarations (E2439)

The "typename" keyword must only be used within template declarations and definitions.

1.1.3.1.109 Cannot generate specialization from 'specifier' because that type is not yet defined (E2440)

The compiler no longer generates this error.

1.1.3.1.110 Instantiating 'specifier' (E2441)

Whenever a compiler error occurs while instantiating a template type, the context of what was being instantiated at that point in time will be reported to the user, in order to aid in detection of the problem.

1.1.3.1.111 Missing or incorrect version of TypeLibImport.dll (E2503)

This error occurs when the compiler is trying to access TypeLibImport.dll but it either can't find it, it was corrupted, or you have the wrong version of it installed on your computer. You can reinstall it from the product CD.

1.1.3.1.112 Need to include header <typeinfo> to use typeid (E2470)

When you use the 'typeid' function, you have to include the <typeinfo> header, otherwise you will get syntax errors.

For example, consider a test case with the following code:

```
int func()
{
  char * name = typeid(int).name();  // This causes an error
}
```

1.1.3.1.113 Cannot (yet) use member overload resolution during template instantiation (E2514)

You are trying to overload a member during template instantiation. You cannot have calls to overloaded constant functions within array bounds initializers, for example.

1.1.3.1.114 'using' cannot refer to a template specialization (E2508)

The `using` keyword cannot refer to a template specialization.

1.1.3.1.115 'virtual' can only be used with non-template member functions (E2462)

The 'virtual' keyword can only be applied to regular member functions, not to member template functions.

Consider a test case with the following code:

```
template <class T>
class myTemplateClass
{
  virtual int func1();                    // This is fine
  template <class T> virtual int func2(); // This causes an error
};
class myClass
{
  virtual int func1();                    // This is fine
  template <class T> virtual int func2(); // This causes an error
};
```

1.1.3.1.116 Informational messages

The compiler displays status information while compiling if you have checked "Show general messages" on the Compiler page of the Project Options dialog box. Most of the messages are self-explanatory and state information about compiling and linking; for example:

```
[C++] Compiling: D:\Program Files\Borland\CBuilder\Bin\Unit1.cpp
```

You may also see a message such as

```
[C++] Including clx.h instead of clx.h due to -Hr switch
```

This message indicates that the IDE-managed precompiled header file was replaced due to additional CLX classes being included in the project. It does not indicate a failure condition. Sometimes the header file name being included may be the same if it is being included with a different #define statement.

1.1.3.1.117 Incorrect use of #pragma alias "aliasName"="substituteName" (W8086)

The directive #pragma alias is used to tell the linker that two identifier names are equivalent. You must put the two names in quotes.

You will receive this warning if you don't use pragma alias correctly. For example, the following two lines both generate this warning:

```
#pragma alias foo=bar
```

```
#pragma alias "foo" = bar
```

See the “#pragma alias” reference below for information on this pragma's usage.

See Also

#pragma alias (see page 156)

1.1.3.1.118 Static main is not treated as an entry point (W8099)

The main function has been created as static, and as such cannot be used as a valid entry point.

Consider:

```
static void main(int argc, char**argv)
{
}
```

The above is valid C syntax, but cannot actually be called as the startup routine since it has been declared static.

1.1.3.1.119 Incorrect use of #pragma codeseg [seg_name] ["seg_class"] [group] (W8093)

The #pragma codeseg directive can be used to set or reset the name, class, and group of a segment. You have to follow the exact syntax mentioned in the warning message, and all names are optional.

So these are all legal:

```
#pragma codeseg
#pragma codeseg foo
#pragma codeseg foo "bar"
#pragma codeseg foo "bar" foobar
```

But these are not:

```
#pragma codeseg ###
#pragma codeseg "foo" "bar"
#pragma codeseg foo "bar" foobar morefoobar
```

1.1.3.1.120 Incorrect use of #pragma comment(<type> [,"string"]) (W8094)

The directive #pragma comment can be used to emit linker comment records.

In this message, <type> can be any of the following:

- user
- lib
- exestr
- linker

The type should be there but the string is optional.

1.1.3.1.121 Function 'function' redefined as non-inline (W8085)

This warning is used to indicate when a certain function, which has been declared inline in one location, is redefined in another location to be non-inline.

1.1.3.1.122 Incorrect use of #pragma message("string") (W8095)

You can use pragma message to emit a message to the command line or to the message window. You would get this warning if you use the incorrect syntax, so

```
#pragma message( "hi there" )
```

is correct, but the following code would generate the warning:

```
#pragma message( badly formed )
```

1.1.3.1.123 Multi-character character constant (W8098)

This warning is issued when the compiler detects a multi-character integer constant, such as:

```
int foo = 'abcd';
```

The problem with this construct is that the byte order of the characters is implementation dependent.

1.1.3.1.124 Incorrect use of #pragma code_seg(["seg_name"[,"seg_class"]]) (W8096)

Pragma code_seg is similar to pragma codeseg, but with this one you can only modify the name and the class of a code segment. If you use the wrong syntax, you get this warning.

The following examples show the correct usage:

```
#pragma code_seg()
#pragma code_seg("foo")
#pragma code_seg("foo", "bar")
```

However, the following incorrect examples will all produce the warning:

```
#pragma code_seg(foo)
#pragma code_seg(foo, bar)
```

1.1.3.1.125 Pragma pack pop with no matching pack push (W8083)

Each #pragma pack(pop) should have a matching preceding #pragma pack(push) in the same translation unit. Pairs of 'push' and 'pop' can be nested.

For example:

```
#pragma pack( push )
#pragma pack( push )
#pragma pack( pop )
#pragma pack( pop )
#pragma pack( pop )   // This causes an error
```

1.1.3.1.126 Not all options can be restored at this time (W8097)

Your program has a #pragma pop at a place where it can't restore options.

For example:

```
#pragma option push -v
int main()
{
  int i;
  i = 1;
#pragma option pop
```

```
  return i;
}
```

For this example, compile with -v-. The message happens because the first #pragma causes debug info to change state (turns it on). Then, in the middle of the function where it is useless to toggle the debug info state, the #pragma pop attempts to return to the former state.

1.1.3.1.127 Suggest parentheses to clarify precedence (W8084)

This warning indicates that several operators used in one expression might cause confusion about the applicable operator precedence rules. The warning helps create code that is more easy to understand and potentially less ambiguous.

For example, compile the following code using the `-w` command line option:

```
int j, k, l;
int main()
{
  return j < k & l;  // This causes an error
}
//
```

1.1.3.1.128 'type' argument 'specifier' passed to 'function' is not an iterator: 'type' iterator required (W8092)

An argument that is not an iterator is being used with an STL algorithm that requires an iterator.

1.1.3.1.129 'operator::operator==' must be publicly visible to be contained by a 'type' (W8087)

A type that is being used with an STL container has a private 'operator=='.

1.1.3.1.130 'type::operator<' must be publicly visible to be used with 'type' (W8090)

A type that is being used with an STL container has a private 'operator<'. The type you're trying to use must be made public.

1.1.3.1.131 'type::operator<' must be publicly visible to be contained by a 'type' (W8089)

The type that is being used for an STL container has a private 'operator<'. The type that is being contained (type::operator) must be a public type.

For example, if you were trying to instantiate a class type "vector<blah>", the error would be:

```
'blah::operator<' must be publicly visible to be contained by a 'vector'
```

1.1.3.1.132 'type' argument 'specifier' passed to 'function' is a 'iterator category' iterator: 'iterator category' iterator required (W8091)

An incorrect iterator category is being used with an STL algorithm.

1.1.3.1.133 Template instance 'specifier' is already instantiated (W8076)

You are trying to explicitly instantiate a template that was already implicitly instantiated.

If –A is not enabled and an attempt is made to explicitly instantiate a specialization which has already been either implicitly or explicitly instantiated, this error will result.

1.1.3.1.134 Explicitly specializing an explicitly specialized class member makes no sense (W8077)

Internal error. This warning is no longer generated by the compiler.

The following code is illegal:

```
template<class T> class foo { int x; }
template<> class foo<int> { int y; }
template<> int foo<int>::y;   // error: cannot explicitly specialize of 'foo<int>::x'
```

1.1.3.1.135 Unable to create output file 'filename' (F1002)

This error occurs if the work disk is full or write protected.

This error also occurs if the output directory does not exist.

Solutions

If the disk is full, try deleting unneeded files and restarting the compilation.

If the disk is write-protected, move the source files to a writeable disk and restart the compilation.

1.1.3.1.136 Error directive: 'message' (F1003)

This message is issued when an `#error` directive is processed in the source file.

'message' is the text of the `#error` directive.

1.1.3.1.137 Internal compiler error (F1004)

An error occurred in the internal logic of the compiler. This error shouldn't occur in practice, but is generated in the event that a more specific error message is not available.

1.1.3.1.138 Include files nested too deep (F1005)

This message flags (directly or indirectly) recursive `#include` directives.

1.1.3.1.139 Bad call of intrinsic function (F1006)

You have used an intrinsic function without supplying a prototype. You may have supplied a prototype for an intrinsic function that was not what the compiler expected.

1.1.3.1.140 Irreducible expression tree (F1007)

An expression on the indicated line of the source file caused the code generator to be unable to generate code. Avoid using the expression. Notify CodeGear if an expression consistently reproduces this error.

1.1.3.1.141 Out of memory (F1008)

The total working storage is exhausted.

This error can occur in the following circumstances:

- Not enough virtual memory is available for compiling a particular file. In this case, shut down any other concurrent applications. You may also try to reconfigure your machine for more available virtual memory, or break up the source file being compiled into smaller separate components. You can also compile the file on a system with more available RAM.
- The compiler has encountered an exceedingly complex or long expression at the line indicated and has insufficient reserves to parse it. Break the expression down into separate statements.

1.1.3.1.142 Unable to open input file 'filename' (F1009)

This error occurs if the source file can't be found.

Check the spelling of the name. Make sure the file is on the specified disk or directory.

Verify that the proper directory paths are listed. If multiple paths are required, use a semicolon to separate them.

1.1.3.1.143 Unable to open 'filename' (F1010)

This error occurs if the specified file can't be opened.

Make sure the file is on the specified disk or directory. Verify the proper paths are listed. If multiple paths are required, use a semicolon to separate them.

1.1.3.1.144 Register allocation failure (F1011)

Possible Causes

An expression on the indicated line of the source file was so complicated that the code generator could not generate code for it.

Solutions

Simplify the expression. If this does not solve the problem, avoid the expression.

Notify CodeGear if an expression can consistently reproduce this error.

1.1.3.1.145 Compiler stack overflow (F1012)

The compiler's stack has overflowed. This can be caused by a number of things, among them deeply nested statements in a function body (for example, if/else) or expressions with a large number of operands. You must simplify your code if this message occurs. Adding more memory to your system will not help.

1.1.3.1.146 Error writing output file (F1013)

A DOS error that prevents the C++ IDE from writing an .OBJ, .EXE, or temporary file.

Solutions

Make sure that the Output directory in the Directories dialog box is a valid directory.

Check that there is enough free disk space.

1.1.3.1.147 Compiler table limit exceeded (F1000)

One of the compiler's internal tables overflowed.

This usually means that the module being compiled contains too many function bodies.

This limitation will not be solved by making more memory available to the compiler. You need to simplify the file being compiled.

1.1.3.1.148 286/287 instructions not enabled (E2000)

Use the -2 command-line compiler option to enable 286/287 opcodes. Be aware that the resulting code cannot be run on 8086- and 8088-based machines.

1.1.3.1.149 Abnormal program termination

The program called abort because there wasn't enough memory to execute.

This message can be caused by memory overwrites.

1.1.3.1.150 Attempt to grant or reduce access to 'identifier' (E2009)

A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class.

It can't add or reduce access rights.

1.1.3.1.151 Illegal to take address of bit field (E2011)

It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.

1.1.3.1.152 Cannot add or subtract relocatable symbols (E2010)

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant.

Variables, procedures, functions, and labels are relocatable symbols.

1.1.3.1.153 Cannot take address of 'main' (E2012)

In C++, it is illegal to take the address of the main function.

1.1.3.1.154 'function1' cannot be distinguished from 'function2' (E2013)

The parameter type lists in the declarations of these two functions do not differ enough to tell them apart.

Try changing the order of parameters or the type of a parameter in one declaration.

1.1.3.1.155 Member is ambiguous: 'member1' and 'member2' (E2014)

You must qualify the member reference with the appropriate base class name.

In C++ class 'class', member 'member' can be found in more than one base class, and it was not qualified to indicate which one you meant.

This applies only in multiple inheritance, where the member name in each base class is not hidden by the same member name in a derived class on the same path.

The C++ language rules require that this test for ambiguity be made before checking for access rights (private, protected, public).

It is possible to get this message even though only one (or none) of the members can be accessed.

1.1.3.1.156 Ambiguity between 'function1' and 'function2' (E2015)

Both of the named overloaded functions could be used with the supplied parameters.

This ambiguity is not allowed.

1.1.3.1.157 Ambiguous override of virtual base member 'base_function': 'derived_function' (E2016)

A virtual function in a virtual base class was overridden with two or more different functions along different paths in the inheritance hierarchy. For example,

```
struct VB
{
virtual f();
};
struct A:virtual VB
{
virtual f();
};
//{
struct B:virtual VB
virtual f();
}
```

1.1.3.1.158 Ambiguous member name 'name' (E2017)

Whenever a structure member name is used in inline assembly, such a name must be unique. (If it is defined in more than one structure, all of the definitions must agree as to its type and offset within the structures). In this case, an ambiguous member name has been used.

For example:

```
struct A
{
   int a;
   int b;
};
...
asm ax,.a;
```

1.1.3.1.159 Cannot throw 'type' -- ambiguous base class 'base' (E2018)

It is not legal to throw a class that contains more than one copy of a (non-virtual) base class.

1.1.3.1.160 'identifier' cannot be declared in an anonymous union (E2019)

The compiler found a declaration for a member function or static member in an anonymous union.

Such unions can only contain data members.

1.1.3.1.161 Global anonymous union not static (E2020)

In C++, a global anonymous union at the file level must be static.

1.1.3.1.162 Array must have at least one element (E2021)

ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed).

An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with malloc.

You can still use this trick, but you must declare the array element to have (at least) one element if you are compiling in strict ANSI mode.

Declarations (as opposed to definitions) of arrays of unknown size are still allowed.

Example

```
char ray[];         /* definition of unknown size -- ILLEGAL */
char ray[0];        /* definition of 0 size -- ILLEGAL */
extern char ray[];  /* declaration of unknown size -- OK */
```

1.1.3.1.163 Array size too large (E2022)

The declared array is larger than 64K and the 'huge' keyword was not used.

If you need an array of this size, either use the 'huge' modifier, like this:

```
int huge array[70000L];  /* Allocate 140000 bytes */
```

or dynamically allocate it with farmalloc( ) or farcalloc( ), like this:

```
int huge *array = (int huge *) farmalloc (sizeof (int) * 70000); ?? Allocate 140,000 bytes
```

1.1.3.1.164 Array of references is not allowed (E2023)

It is illegal to have an array of references, because pointers to references are not allowed and array names are coerced into pointers.

1.1.3.1.165 Cannot modify a const object (E2024)

This indicates an illegal operation on an object declared to be const, such as an assignment to the object.

1.1.3.1.166 Assignment to 'this' not allowed, use X::operator new instead (E2025)

In early versions of C++, the only way to control allocation of class of objects was by assigning to the 'this' parameter inside a constructor.

This practice is no longer allowed, because a better, safer, and more general technique is to define a member function operator new instead.

For example:

```
this = malloc(n);
```

1.1.3.1.167 Assembler statement too long (E2026)

Inline assembly statements can't be longer than 480 bytes.

1.1.3.1.168 Only __fastcall functions allowed in __automated section (E2002)

The calling convention for functions declared in an __automated section must be __fastcall.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
int__cdecl    barInt(int);// Error
};
```

1.1.3.1.169 Constructors and destructors not allowed in __automated section (E2001)

Only member function declarations are allowed in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
clxclass()  {}// Error
};
```

1.1.3.1.170 Data member definition not allowed in __automated section (E2003)

Only member function declarations are allowed in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
intmemInt;// Error
};
```

1.1.3.1.171 Only read or write clause allowed in property declaration in __automated section (E2004)

Storage specifiers stored, default, and nodefault are not allowed in property declarations in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
int __fastcall Get(void);
__automated:
int __property ip1 = { read = Get };      // OK
int __property ip2 = { read = Get, default = 42 }; // Error
};
```

1.1.3.1.172 Redeclaration of property not allowed in __automated section (E2005)

If you declare a property in an __automated section it has be a new declaration. Property hoisting is not allowed.

Example

```
struct__declspec(delphiclass) clxbaseclass
{
int __fastcall Get(void);
void __fastcall Set(int);
int __property ip1 = { read = Get, write = Set };
};
structclxderivedclass : clxbaseclass
{
int __fastcall NewGetter(void);
__automated:
__property ip1;// Error
int __property ip2 = { read = Get, write = Set }; // OK
};
```

1.1.3.1.173 Must take address of a memory location (E2027)

Your source file used the address-of operator (&) with an expression that can't be used that way; for example, a register variable.

1.1.3.1.174 operator -> must return a pointer or a class (E2028)

The C++ operator -> function must be declared to either return a class or a pointer to a class (or struct or union).

In either case, it must be something to which the -> operator can be applied.

1.1.3.1.175 'identifier' must be a previously defined class or struct (E2029)

You are attempting to declare 'identifier' to be a base class, but either it is not a class or it has not yet been fully defined.

Correct the name or rearrange the declarations.

1.1.3.1.176 Misplaced break (E2030)

The compiler encountered a break statement outside a switch or looping construct.

You can only use break statements inside of switch statements or loops.

1.1.3.1.177 Cannot cast from 'type1' to 'type2' (E2031)

A cast from type 'ident1' to type 'ident2' is not allowed.

In C++, you cannot cast a member function pointer to a normal function pointer.

For example:

```
class A {
public:
   int myex();
};
typedef int (*fp)();
test()
{
   fp myfp = (fp) &A::myex; //error
   return myfp();
}
```

The reason being that a class member function takes a hidden parameter, the this pointer, thus it behaves very differently than a normal function pointer.

A static member function behaves as normal function pointer and can be cast.

For example:

```
class A {
public:
   static int myex();
};
typedef int (*fp)();
test()
{
   fp myfp = (fp) &A::myex; //ok
   return myfp();
}
```

However, static member functions can only access static data members of the class.

In C

- A pointer can be cast to an integral type or to another pointer.
- An integral type can be cast to any integral, floating, or pointer type.
- A floating type can be cast to an integral or floating type.

Structures and arrays can't be cast to or from.

You usually can't cast from a void type.

In C++

User-defined conversions and constructors are checked for. If one can't be found, the preceding rules apply (except for pointers to class members).

Among integral types, only a constant zero can be cast to a member pointer.

A member pointer can be cast to an integral type or to a similar member pointer.

A similar member pointer points to a data member (or to a function) if the original does. The qualifying class of the type being cast to must be the same as (or a base class of) the original.

1.1.3.1.178 Illegal use of closure pointer (E2032)

A closure pointer variable is used incorrectly. Closure variables have limited usage. For instance, you can assign a function to a closure variable, and execute that function through the closure variable, but you cannot use a closure variable like a pointer variable.

1.1.3.1.179 Misplaced continue (E2033)

The compiler encountered a continue statement outside a looping construct.

1.1.3.1.180 Cannot convert 'type1' to 'type2' (E2034)

An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not legal.

In C++, the compiler will convert one function pointer to another only if the signature for the functions are the same. Signature refers to the arguments and return type of the function. For example:

```
myex( int );
typedef int ( *ffp )( float );
test()
{
   ffp fp = myex; //error
}
```

Seeing that myex takes an int for its argument, and fp is a pointer to a function which takes a float as argument, the compiler will not convert it for you.

In cases where this is what is intended, performing a typecast is necessary:

```
myex( int );
typedef int ( *ffp )( float );
test()
{
   ffp fp = (ffp)myex;  //ok
}
```

1.1.3.1.181 Conversion operator cannot have a return type specification (E2036)

This C++ type conversion member function specifies a return type different from the type itself.

A declaration for conversion function operator can't specify any return type.

1.1.3.1.182 The constructor 'constructor' is not allowed (E2037)

Constructors of the form

```
  X(X);  //{ Error
};
```

are not allowed.

This is the correct way to write a copy constructor:

```
class X {
  X(const X&);// Copy constructor
```

```
};
```

1.1.3.1.183 Declaration terminated incorrectly (E2040)

A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body.

A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.

1.1.3.1.184 Misplaced decimal point (E2039)

The compiler encountered a decimal point in a floating-point constant as part of the exponent.

1.1.3.1.185 Incorrect use of default (E2041)

The compiler found no colon after the default keyword.

1.1.3.1.186 Declare operator delete (void*) or (void*, size_t) (E2042)

E2043 Declare operator delete[] (void) or (void*, size_t)*Compiler error

Declare the operator delete with one of the following:

1.A single void* parameter, or

2.A second parameter of type size_t

If you use the second version, it will be used in preference to the first version.

The global operator delete can only be declared using the single-parameter form.

1.1.3.1.187 operator delete must return void (E2044)

*E2044 operator delete[] must return void*Compiler error

This C++ overloaded operator delete was declared in some other way.

Declare the operator delete with one of the following:

1.A single void* parameter, or

2.A second parameter of type size_t

If you use the second version, it will be used in preference to the first version.

The global operator delete can only be declared using the single-parameter form.

1.1.3.1.188 Destructor name must match the class name (E2045)

In a C++ class, the tilde (~) introduces a declaration for the class destructor.

The name of the destructor must be same as the class name.

In your source file, the ~ preceded some other name.

1.1.3.1.189 Unknown preprocessor directive: 'identifier' (E2048)

The compiler encountered a # character at the beginning of a line. The directive name that followed the # was not one of the following:

- define
- else
- endif
- if
- ifdef
- ifndef
- include
- line
- pragma
- undef

1.1.3.1.190 Bad file name format in include directive OR Bad file name format in line directive (E2046)

Include and line directive file names must be surrounded by quotes ("filename.h") or angle brackets (<filename.h>).

The file name was missing the opening quote or angle bracket.

If a macro was used, the resulting expansion text is not surrounded by quote marks.

1.1.3.1.191 Bad 'directive' directive syntax (E2047)

A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name.

An example of this might be:

```
Bad ifdef directive syntax
```

Note that an `#ifdef` directive must contain a single identifier (and nothing else) as the body of the directive.

Another example is:

```
Bad undef directive syntax
```

An `#undef` directive must also contain only one identifier as the body of the directive.

1.1.3.1.192 Class type 'type' cannot be marked as __declspec(delphireturn) (E2049)

Classes marked as delphireturn are special classes that the compiler needs to recognize by name. These classes are predefined in the headers.

Some of the delphireturn classes are Variant, AnsiString, and Currency.

You cannot mark user-defined classes as delphireturn.

1.1.3.1.193 Invalid use of dot (E2051)

An identifier must immediately follow a period operator (.). This is a rare message that can only occur in some specialized inline assembly statements.

Example

```
struct foo {
  int x;
  int y;
}p = {0,0};
int y;
int main (void)
{
  asm mov eax.(foo)x, 1;
  asm mov eax.(foo)4, 1;       /* Error: Invalid use of dot */
  return 0;
}
```

1.1.3.1.194 Dynamic function 'function' conflicts with base class 'class' (E2052)

Some of the modifiers of this dynamic function conflict with the definition of the same function in the base class. The two functions should have the same modifiers. The following modifiers (among others) can cause conflicts:

- __export
- __import
- declspec(naked)
- declspec(package)
- __fastcall

1.1.3.1.195 Misplaced elif directive (E2053)

The compiler encountered an `#elif` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.

1.1.3.1.196 Misplaced else (E2054)

The compiler encountered an `else` statement without a matching if statement.

Possible Causes

- An extra "else" statement
- An extra semicolon
- Missing braces
- Some syntax error in a previous "if" statement

1.1.3.1.197 Misplaced else directive (E2055)

The compiler encountered an `#else` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.

1.1.3.1.198 Misplaced endif directive (E2056)

The compiler encountered an `#endif` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.

1.1.3.1.199 Exception specification not allowed here (E2057)

Function pointer type declarations are not allowed to contain exception specifications.

1.1.3.1.200 Exception handling variable may not be used here (E2058)

An attempt has been made to use one of the exception handling values that are restricted to particular exception handling constructs, such as GetExceptionCode().

1.1.3.1.201 Unknown language, must be C or C++ (E2059)

In the C++ construction

```
extern "name" type func( /*...*/ );
```

the given "name" must be "C" or "C++" (use the quotes); other language names are not recognized.

You can declare an external Pascal function without the compiler's renaming like this:

```
extern "C" int pascal func( /*...*/ );
```

To declare a (possibly overloaded) C++ function as Pascal and allow the usual compiler renaming (to allow overloading), you can do this:

```
extern int pascal func( /*...*/ );
```

1.1.3.1.202 Illegal use of floating point (E2060)

Floating-point operands are not allowed in these operators

- shift (SHL, SHR)
- bitwise Boolean (AND, OR, XOR, NOT)
- conditional (? :)
- indirection (*)
- certain others

The compiler found a floating-point operand with one of these prohibited operators.

1.1.3.1.203 Friends must be functions or classes (E2061)

A friend of a C++ class must be a function or another class.

1.1.3.1.204 Invalid indirection (E2062)

The indirection operator (*) requires a pointer as the operand.

Example

```
int main (void)
```

```
{
  int p;
  *p = 10;     /* ERROR: Invalid Indirection */
  return 0;
}
```

1.1.3.1.205 Illegal initialization (E2063)

Initializations must be one of the following:

- constant expressions
- the address of a global extern or static variable plus or minus a constant

1.1.3.1.206 Cannot initialize 'type1' with 'type2' (E2064)

You are attempting to initialize an object of type 'type1' with a value of type 'type2' which is not allowed.

The rules for initialization are essentially the same as for assignment.

1.1.3.1.207 Using namespace symbol 'symbol' conflicts with intrinsic of the same name (E2065)

If you define a function in a namespace, which has a name that might be replaced by a call to an intrinsic when -Oi is on, it is not permitted to have a "using" declaration which refers to that member.

For example, calls to "strcmp" are replaced by the intrinsic "__strcmp__" when -Oi is on. This means that the declaration "using N::strcmp;" would become "using N::__strcmp__", since the token replacement happens before the compiler's parser ever sees the tokens.

An error displays in this case, because the compiler doesn't know how to process "N::__strcmp__".

1.1.3.1.208 'main' must have a return type of int (E2067)

In C++, function main has special requirements, one of which is that it cannot be declared with any return type other than int.

1.1.3.1.209 'identifier' is not a non-static data member and can't be initialized here (E2068)

Only data members can be initialized in the initializers of a constructor.

This message means that the list includes a static member or function member.

Static members must be initialized outside of the class, for example:

```
class A { static int i; };
int A::i = -1;
```

1.1.3.1.210 Illegal use of member pointer (E2069)

Pointers to class members can only be passed as arguments to functions, or used with the following operators:

- assignment
- comparison

.

->

- ?:
- &&
- ||

The compiler has encountered a member pointer being used with a different operator.

In order to call a member function pointer, one must supply an instance of the class for it to call upon.

For example:

```
class A {
public:
   myex();
};
typedef int (A::*Amfptr)();
myex()
{
   Amfptr mmyex = &A::myex;
   return (*mmyex)();  //error
}
```

This will compile:

```
cl_ass A {
public:
   myex();
};
typedef int (A::*Amfptr)();
foo()
{
   A a;
   Amfptr mmyex = &A::myex;
   return (a.*mmyex)();
}
```

1.1.3.1.211 operator new must have an initial parameter of type size_t (E2071)

*E2071 Operator new[] must have an initial parameter of type size_t*Compiler error

Operator new can be declared with an arbitrary number of parameters.

It must always have at least one, the amount of space to allocate.

1.1.3.1.212 Operator new[] must return an object of type void (E2072)

This C++ overloaded operator new was declared in some other way.

1.1.3.1.213 Incorrect 'type' option: option (E2075)

An error has occurred in either the configuration file or a command-line option. The compiler may not have recognized the configuration file parameter as legal; check for a preceding hyphen (-), or the compiler may not have recognized the command-line parameter as legal.

This error can also occur if you use a `#pragma` option in your code with an invalid option.

1.1.3.1.214 Nothing allowed after pragma option pop (E2073)

The #pragma option pop can only be followed by comments, blanks, or end of line.

1.1.3.1.215 Overloadable operator expected (E2076)

Almost all C++ operators can be overloaded.

These are the only ones that can't be overloaded:

- the field-selection dot (`.`)
- dot-star (`.*`)
- double colon (`::`)
- conditional expression (`?:`)

The preprocessor operators (# and ##) are not C or C++ language operators and thus can't be overloaded.

Other non-operator punctuation, such as semicolon (;), can't be overloaded.

1.1.3.1.216 'function' must be declared with one parameter (E2080)

This C++ operator function was incorrectly declared with more than one parameter.

1.1.3.1.217 'operator' must be declared with one or no parameters (E2077)

When operator ++ or operator -- is declared as a member function, it must be declared to take either:

- No parameters (for the prefix version of the operator), or
- One parameter of type int (for the postfix version)

1.1.3.1.218 'function' must be declared with no parameters (E2079)

This C++ operator function was incorrectly declared with parameters.

1.1.3.1.219 'operator' must be declared with one or two parameters (E2078)

When operator ++ or operator -- is declared as a non-member function, it must be declared to take either:

- one parameter (for the prefix version of the operator), or
- two parameters (for the postfix version)

1.1.3.1.220 'function' must be declared with two parameters (E2081)

This C++ operator function was incorrectly declared with other than two parameters.

1.1.3.1.221 'identifier' must be a member function or have a parameter of class type (E2082)

Most C++ operator functions must have an implicit or explicit parameter of class type.

This operator function was declared outside a class and does not have an explicit parameter of class type.

1.1.3.1.222 Last parameter of 'operator' must have type 'int' (E2083)

When a postfix operator ++ or operator -- is overloaded, the last parameter must be declared with the type int.

1.1.3.1.223 Parameter names are used only with a function body (E2084)

When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype.

A list of parameter names only is not allowed.

Example declarations

```
int func();             /* declaration without prototype -- OK */
int func(int, int);     /* declaration with prototype -- OK */
int func(int i, int j); /* parameter names in prototype -- OK */
int func(i, j);         /* parameter names only -- ILLEGAL */
```

1.1.3.1.224 Invalid pointer addition (E2085)

Your source file attempted to add two pointers together.

1.1.3.1.225 Illegal pointer subtraction (E2086)

This is caused by attempting to subtract a pointer from a non-pointer.

1.1.3.1.226 Illegal use of pointer (E2087)

Pointers can only be used with these operators:

- addition(+)
- subtraction(-)
- assignment(=)
- comparison(==)
- indirection(*)
- arrow(->)

Your source file used a pointer with some other operator.

Example

```
int main (void)
{
  char *p;
  p /= 7;     /* ERROR: Illegal Use of Pointer */
  return 0;
}
```

1.1.3.1.227 Bad syntax for pure function definition (E2088)

Pure virtual functions are specified by appending "= 0" to the declaration, like this:

```
class A { virtual void f () = 0;}
class B : public A { void f () {}; }
```

You wrote something similar, but it was not correct.

1.1.3.1.228 Identifier 'identifier' cannot have a type qualifier (E2089)

A C++ qualifier class::identifier can't be applied here.

A qualifier is not allowed on the following:

- typedef names
- function declarations (except definitions at the file level)
- on local variables or parameters of functions
- on a class member--except to use its own class as a qualifier (redundant but legal).

1.1.3.1.229 Qualifier 'identifier' is not a class or namespace name (E2090)

The C++ qualifier in the construction qual::identifier is not the name of a struct or class.

1.1.3.1.230 Functions cannot return arrays or functions (E2091)

A function was defined to return an array or a function. Check to see if either the intended return was a pointer to an array or function (and perhaps the * is missing) or if the function definition contained a request for an incorrect datatype.

1.1.3.1.231 Storage class 'storage class' is not allowed here (E2092)

The given storage class is not allowed here.

Probably two storage classes were specified, and only one can be given.

1.1.3.1.232 Operator 'operator' not implemented in type 'type' for arguments of the same type (E2093)

The operator you are calling is not defined in this class. When you have an expression: x + x, where x is of type class X, the operator + has to be defined in class X and be accessible.

1.1.3.1.233 Operator 'operator' not implemented in type 'type' for arguments of type 'type' (E2094)

The operator you are calling is not defined in this class. When you have an expression: x + x, where x is of type class X, the operator + has to be defined in class X and be accessible.

1.1.3.1.234 Illegal structure operation (E2096)

Structures can only be used with dot (.), address-of (&) or assignment (=) operators, or be passed to or from a function as parameters.

The compiler encountered a structure being used with some other operator.

1.1.3.1.235 Explicit instantiation only allowed at file or namespace scope (E2097)

The explicit instantiation operator "template" can only be used within global or namespace scope. It cannot be used to qualify a local class or a class member, for example.

1.1.3.1.236 'export' keyword must precede a template declaration (E2101)

The 'export' keyword can only occur before the keyword "template" ina template declaration. It cannot be used anywhere else.

1.1.3.1.237 Explicit instantiation must be used with a template class or function (E2103)

The explicit instantiation operator "template" can only be used to refer to templates. It cannot be used with non-templates.

1.1.3.1.238 Explicit specialization must be used with a template class or function (E2106)

The explicit specialization operator template<> can only be used in front of a template class or function. Using it with a normal class means nothing, and hence generates an error.

1.1.3.1.239 Invalid use of template keyword (E2104)

You can only use a template class name without specifying its actual arguments inside a template definition.

Using a template class name without specifying its actual arguments outside a template definition is illegal.

1.1.3.1.240 Explicit specialization declarator "template<>" now required (E2098)

When specializing a function, such as providing the definition for "foo<int>", so that foo behaves specially which called for the "int" argument, now requires that the declaration begin with an explicit specialization operator.

1.1.3.1.241 Improper use of typedef 'identifier' (E2108)

Your source file used a typedef symbol where a variable should appear in an expression.

Check for the declaration of the symbol and possible misspellings.

1.1.3.1.242 Explicit specialization only allowed at file or namespace scope (E2099)

The explicit specialization operator template<> can only be used within global or namespace scope. It cannot be used to qualify a local class or a class member, for example.

1.1.3.1.243 Not an allowed type (E2109)

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

1.1.3.1.244 Incompatible type conversion (E2110)

The cast requested can't be done.

1.1.3.1.245 Type 'typename' may not be defined here (E2111)

Class and enumeration types may not be defined in a function return type, a function argument type, a conversion operator type, or the type specified in a cast.

You must define the given type before using it in one of these contexts.

`Note:`This error message is often the result of a missing semicolon (`;`) for a class declaration. You might want to verify that all the class declarations preceding the line on which the error occurred end with a semicolon.

1.1.3.1.246 Unknown unit directive: 'directive' (E2112)

You cannot use this name as a unit directive. Instead use one of the following unit directives: weak, smart_init, or deny.

1.1.3.1.247 Virtual function 'function1' conflicts with base class 'base' (E2113)

A virtual function has the same argument types as one in a base class, but differs in one or more of the following:

- Return type
- Calling convention
- Exception specification (throw list)

1.1.3.1.248 Multiple base classes require explicit class names (E2114)

In a C++ class constructor, if there is more than one immediate base class, each base class constructor call in the constructor header must include the base class name.

1.1.3.1.249 Bit field too large (E2115)

This error occurs when you supply a bit field with more than 16 bits.

1.1.3.1.250 Bit fields must contain at least one bit (E2116)

You can't declare a named bit field to have 0 (or less than 0) bits.

You can declare an unnamed bit field to have 0 bits.

This is a convention used to force alignment of the following bit field to a byte boundary (or to a word boundary.

1.1.3.1.251 Bit fields must be signed or unsigned int (W8005)

In ANSI C, bit fields may only be signed or unsigned int (not char or long, for example).

1.1.3.1.252 Bit fields must have integral type (E2118)

In C++, bit fields must have an integral type. This includes enumerations.

1.1.3.1.253 User break (E2119)

You typed a Ctrl+Break while compiling in the IDE.

(This is not an error, just a confirmation.)

1.1.3.1.254 Cannot call 'main' from within the program (E2120)

C++ does not allow recursive calls of main( ).

1.1.3.1.255 Function call missing) (E2121)

The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

1.1.3.1.256 Cannot declare or define 'identifier' here: wrong namespace (E2038)

You tried to declare a template in an illegal place or a namespace member outside of its namespace.

1.1.3.1.257 Cannot define 'identifier' using a namespace alias (E2154)

You cannot use a namespace alias to define a namespace member outside of its namespace.

1.1.3.1.258 Cannot use local type 'identifier' as template argument (E2421)

A local type was used in an actual template type argument, which is illegal.

1.1.3.1.259 Class 'class' may not contain pure functions (E2123)

The class being declared cannot be abstract, and therefore it cannot contain any pure functions.

1.1.3.1.260 Compiler could not generate copy constructor for class 'class' OR Compiler could not generate default constructor for class 'class' OR Compiler could not generate operator = for class 'class' (E2125)

Sometimes the compiler is required to generate a member function for the user.

Whenever such a member function can't be generated due to applicable language rules, the compiler issues one of these error

messages.

1.1.3.1.261 Case bypasses initialization of a local variable (E2126)

In C++ it is illegal to bypass the initialization of a local variable.

This error indicates a case label that can transfer control past this local variable.

1.1.3.1.262 Case statement missing : (E2127)

A case statement must have a constant expression followed by a colon.

The expression in the case statement either was missing a colon or had an extra symbol before the colon.

1.1.3.1.263 Case outside of switch (E2128)

The compiler encountered a case statement outside a switch statement.

This is often caused by mismatched braces.

1.1.3.1.264 Character constant too long (or empty) (E2129)

Character constants can only be one or two characters long.

1.1.3.1.265 Circular property definition (E2130)

Indicates that a property definition relies directly or indirectly on itself.

Example

```
struct pbase
{
     int __property ip1 = {read = ip2, write = ip2};
     int __property ip2 = {read = ip1, write = ip1};
};
```

The above code sample will cause this error message on any usage of ip1 or ip2.

1.1.3.1.266 Objects of type 'type' cannot be initialized with { } (E2131)

Ordinary C structures can be initialized with a set of values inside braces.

C++ classes can only be initialized with constructors if the class has constructors, private members, functions, or base classes that are virtual.

1.1.3.1.267 Unable to execute command 'command' (E2133)

The linker or assembler cannot be found, or possibly the disk is bad.

1.1.3.1.268 Null pointer assignment

When a small or medium memory model program exits, a check is made to determine if the contents of the first few bytes within the program's data segment have changed. These bytes would never be altered by a working program. If they have been

changed, this message is displayed to inform you that (most likely) a value was stored to an uninitialized pointer.

The program might appear to work properly in all other respects; however, this is a serious bug which should be attended to immediately. Failure to correct an uninitialized pointer can lead to unpredictable behavior (including locking the computer up in the large, compact, and huge memory models).

You can use the integrated debugger to track down null pointers.

1.1.3.1.269 Compound statement missing closing brace (E2134)

The compiler reached the end of the source file and found no closing brace.

This is most commonly caused by mismatched braces.

1.1.3.1.270 Call to undefined function 'function' (E2268)

Your source file declared the current function to return some type other than `void` in C++ (or `int` in C), but the compiler encountered a return with no value. All `int` functions are exempt in C because in old versions of C, there was no `void` type to indicate functions that return nothing.

1.1.3.1.271 Destructor for 'class' required in conditional expression (E2137)

If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor because the variable might or might not have been initialized.

The temporary can be explicitly created, as with classname(val, val), or implicitly created by some other code.

You should recast your code to eliminate this temporary value.

1.1.3.1.272 Constructor/Destructor cannot be declared 'const' or 'volatile' (E2135)

A constructor or destructor has been declared as `const` or `volatile`.

This is not allowed.

1.1.3.1.273 Conflicting type modifiers (E2138)

This occurs when a declaration is given that includes more than one addressing modifier on a pointer or more than one language modifier for a function.

Only one language modifier (for example, `__cdecl`, `__pascal`, or `__fastcall`) can be given for a function.

1.1.3.1.274 Constructor cannot have a return type specification (E2136)

C++ constructors have an implicit return type used by the compiler, but you can't declare a return type or return a value.

1.1.3.1.275 Conversions of class to itself or base class not allowed (E2035)

You tried to define a conversion operator to the same class or a base class.

1.1.3.1.276 Declaration missing ; (E2139)

Your source file contained a struct or union field declaration that was not followed by a semicolon.

Check previous lines for a missing semicolon.

1.1.3.1.277 Declaration is not allowed here (E2140)

Declarations can't be used as the control statement for while, for, do, if, or switch statements.

1.1.3.1.278 Declaration syntax error (E2141)

Your source file contained a declaration that was missing a symbol or had an extra symbol added to it.

Check for a missing semicolon or parenthesis on that line or on previous lines.

1.1.3.1.279 Base class 'class' contains dynamically dispatchable functions (E2142)

This error occurs when a class containing a DDVT function attempts to inherit DDVT functions from multiple parent classes.

Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance.

1.1.3.1.280 Matching base class function 'function' has different dispatch number (E2143)

If a DDVT function is declared in a derived class, the matching base class function must have the same dispatch number as the derived function.

1.1.3.1.281 Matching base class function 'function' is not dynamic (E2144)

If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.

1.1.3.1.282 Functions 'function1' and 'function2' both use the same dispatch number (E2145)

This error indicates a dynamically dispatched virtual table (DDVT) problem.

1.1.3.1.283 Need an identifier to declare (E2146)

In this context, an identifier was expected to complete the declaration.

This might be a typedef with no name, or an extra semicolon at file level.

In C++, it might be a class name improperly used as another kind of identifier.

1.1.3.1.284 'identifier' cannot start a parameter declaration (E2147)

An undefined 'identifier' was found at the start of an argument in a function declarator.

Often the type name is misspelled or the type declaration is missing. This is usually caused by not including the appropriate header file.

1.1.3.1.285 Default argument value redeclared (E2149)

When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.

1.1.3.1.286 Default argument value redeclared for parameter 'parameter' (E2148)

When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.

1.1.3.1.287 Type mismatch in default argument value (E2150)

The default parameter value given could not be converted to the type of the parameter.

The message "Type mismatch in default argument value" is used when the parameter was not given a name.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is most often "Cannot convert 'type1' to 'type2'" but the mismatch could be due to another reason.

1.1.3.1.288 Type mismatch in default value for parameter 'parameter' (E2151)

The default parameter value given could not be converted to the type of the parameter.

The message "Type mismatch in default argument value" is used when the parameter was not given a name.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.289 Default expression may not use local variables (E2152)

A default argument expression is not allowed to use any local variables or other parameters.

1.1.3.1.290 Define directive needs an identifier (E2153)

The first non-whitespace character after a `#define` must be an identifier.

The compiler found some other character.

1.1.3.1.291 Too many default cases (E2155)

The compiler encountered more than one default statement in a single switch.

1.1.3.1.292 Default outside of switch (E2156)

The compiler encountered a default statement outside a switch statement.

This is most commonly caused by mismatched braces.

1.1.3.1.293 Deleting an object requires exactly one conversion to pointer operator (E2157)

If a person uses the 'delete' operator on an object (note: not a pointer to an object, but an object itself), the standard requires that object to define exactly one "conversion to pointer operator" which will yield the pointer that gets freed. For example:

```
char *a = new char[10];
class foo {
public:
    operator char *() { return a; }
};

int main() {
    delete[] x;
}
```

Since 'x' is not a pointer, but an object, the compiler will delete 'a', because that is what the pointer conversion operator for the object yields. Having more than one conversion to pointer operator is illegal, because the compiler would not know which one to call.

1.1.3.1.294 Operand of 'delete' must be non-const pointer (E2158)

It is illegal to delete a variable that is not a pointer. It is also illegal to delete a pointer to a constant.

For example:

```
const int x=10;
   const int * a = &x;
   int * const b = new int;
   int &c = *b;
   delete a;   //illegal - deleting pointer to constant
   delete b;   //legal
   delete c;   //illegal - operand not of pointer type
               //should use 'delete&c' instead
```

1.1.3.1.295 Trying to derive a far class from the huge base 'base' (E2159)

This error is no longer generated by the compiler.

1.1.3.1.296 Trying to derive a far class from the near base 'base' (E2160)

If a class is declared (or defaults to) near, all derived classes must also be near.

1.1.3.1.297 Trying to derive a huge class from the far base 'base' (E2161)

This error is no longer generated by the compiler.

1.1.3.1.298 Trying to derive a huge class from the near base 'base' (E2162)

This error is no longer generated by the compiler.

1.1.3.1.299 Trying to derive a near class from the far base 'base' (E2163)

If a class is declared (or defaults to) far, all derived classes must also be far.

1.1.3.1.300 Trying to derive a near class from the huge base 'base' (E2164)

This error is no longer generated by the compiler.

1.1.3.1.301 Destructor cannot have a return type specification (E2165)

C++ destructors never return a value, and you can't declare a return type or return a value.

1.1.3.1.302 Destructor for 'class' is not accessible (E2166)

The destructor for this C++ class is protected or private, and can't be accessed here to destroy the class.

If a class destructor is private, the class can't be destroyed, and thus can never be used. This is probably an error.

A protected destructor can be accessed only from derived classes.

This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.

1.1.3.1.303 'function' was previously declared with the language 'language' (E2167)

Only one language modifier (cdecl pascal) can be given for a function.

This function has been declared with different language modifiers in two locations.

1.1.3.1.304 Dispid only allowed in __automated sections (E2007)

The definition of dispids is only permitted in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
int __fastcall foo1(void) __dispid(42);// Error
__automated:
int __fastcall foo2(void) __dispid(43);// OK
};
```

1.1.3.1.305 Divide error

You tried to divide an integer by zero, which is illegal.

1.1.3.1.306 Division by zero (E2168)

Your source file contains a divide or remainder in a constant expression with a zero divisor.

1.1.3.1.307 'identifier' specifies multiple or duplicate access (E2169)

A base class can be declared public or private, but not both.

This access specifier can appear no more than once for a base class.

1.1.3.1.308 Base class 'class' is included more than once (E2170)

A C++ class can be derived from any number of base classes, but can be directly derived from a given class only once.

1.1.3.1.309 Body has already been defined for function 'function' (E2171)

A function with this name and type was previously supplied a function body.

A function body can only be supplied once.

One cause of this error is not declaring a default constructor which you implement. For example:

```
class A {
public:
   virtual myex();
};
A::A() {} // error
```

Having not seen you declare the default constructor in the class declaration, the compiler has had to generate one, thus giving the error message when it sees one. this is a correct example:

```
class A {
public:
   A();
   virtual myex();
};
A::A() {}
```

1.1.3.1.310 Duplicate case (E2172)

Each case of a switch statement must have a unique constant expression value.

1.1.3.1.311 Duplicate handler for 'type1', already had 'type2' (E2173)

It is not legal to specify two handlers for the same type.

1.1.3.1.312 The name handler must be last (E2174)

In a list of catch handlers, if the specified handler is present, it must be the last handler in the list (that is, it cannot be followed by

any more catch handlers).

1.1.3.1.313 Dispid number already used by identifier (E2180)

Dispids must be unique and the compiler checks for this.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int __fastcall foo1(void) __dispid(42);// OK
int __fastcall foo2(void) __dispid(42);// Error
};
```

1.1.3.1.314 Too many storage classes in declaration (E2175)

A declaration can never have more than one storage class, either Auto, Register, Static, or Extern.

1.1.3.1.315 Too many types in declaration (E2176)

A declaration can never have more than one basic type. Examples of basic types are:

- char
- class
- int
- float
- double
- struct
- union
- enum
- typedef name

1.1.3.1.316 Redeclaration of #pragma package with different arguments (E2177)

You can have multiple #pragma package statements in a source file as long as they have the same arguments. This error occurs if the compiler encounters more than one #pragma package with different arguments in each.

1.1.3.1.317 VIRDEF name conflict for 'function' (E2178)

The compiler must truncate mangled names to a certain length because of a name length limit that is imposed by the linker. This truncation may (in very rare cases) cause two names to mangle to the same linker name. If these names happen to both be VIRDEF names, the compiler issues this error message. The simplest workaround for this problem is to change the name of 'function' so that the conflict is avoided.

1.1.3.1.318 virtual specified more than once (E2179)

The C++ reserved word "virtual" can appear only once in one member function declaration.

1.1.3.1.319 Cannot override a 'dynamic/virtual' with a 'dynamic/virtual' function (E2181)

When you declare a function dynamic, you cannot override this function in a derived class with a virtual function of the same name and type. Similarly when the function is declared virtual, you cannot override it with a dynamic one in a derived class.

1.1.3.1.320 Earlier declaration of 'identifier' (E2344)

This error message only shows up after the messages "Multiple declaration for 'identifier'" and "Type mismatch in redeclaration of 'identifier'". It tells you where the previous definition of the identifier in question was found by the compiler, so you don't have to search for it.

1.1.3.1.321 Illegal parameter to __emit__ (E2182)

There are some restrictions on inserting literal values directly into your code with the __emit__ function.

For example, you cannot give a local variable as a parameter to __emit__.

1.1.3.1.322 File must contain at least one external declaration (E2183)

This compilation unit was logically empty, containing no external declarations.

ANSI C and C++ require that something be declared in the compilation unit.

1.1.3.1.323 Enum syntax error (E2184)

An `enum` declaration did not contain a properly formed list of identifiers.

1.1.3.1.324 The value for 'identifier' is not within the range of an int (E2185)

All enumerators must have values that can be represented as an integer.

You have attempted to assign a value that is out of the range of an integer.

If you need a constant of this value, use a const integer.

1.1.3.1.325 Unexpected end of file in comment started on line 'number' (E2186)

The source file ended in the middle of a comment.

This is normally caused by a missing close of comment (*/).

1.1.3.1.326 Unexpected end of file in conditional started on line 'number' (E2187)

The source file ended before the compiler (or MAKE) encountered `#endif`.

The `#endif` either was missing or misspelled.

Every `#if` statement needs a matching `#endif` statement.

1.1.3.1.327 Expression syntax (E2188)

This is a catch-all error message when the compiler parses an expression and encounters a serious error.

Possible Causes

This is most commonly caused by one of the following:

- two consecutive operators
- mismatched or missing parentheses
- a missing semicolon on the previous statement.

Solutions

If the line where the error occurred looks syntactically correct, look at the line directly above for errors.

Try moving the line with the error to a different location in the file and recompiling.

If the error still occurs at the moved statement, the syntax error is occurring somewhere in that statement.

If the error occurred in another statement, the syntax error is probably in the surrounding code.

1.1.3.1.328 Unexpected closing brace (E2190)

An extra right brace was encountered where none was expected. Check for a missing closing brace.

Useful Tip:

The IDE has a mechanism for finding a matching curly brace. If you put the cursor on the '{' or '}' character, hold down Ctrl, hit 'Q' and then '{' or '}', it will position the cursor on the matching brace.

1.1.3.1.329 extern variable cannot be initialized (E2189)

The storage class extern applied to a variable means that the variable is being declared but not defined here--no storage is being allocated for it.

Therefore, you can't initialize the variable as part of the declaration.

1.1.3.1.330 Too few parameters in call (E2192)

This error message occurs when a call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given. Make certain that your call to a function has the same parameters as the function prototype.

1.1.3.1.331 Too few parameters in call to 'function' (E2193)

A call to the named function (declared using a prototype) has too few arguments.

Make certain that the parameters in the call to the function match the parameters of the function prototype.

1.1.3.1.332 Could not find file 'filename' (E2194)

The compiler is unable to find the file supplied on the command line.

1.1.3.1.333 Floating point error: Divide by 0 OR Floating point error: Domain OR Floating point error: Overflow

These fatal errors result from a floating-point operation for which the result is not finite:

- Divide by 0 means the result is +INF or -INF exactly, such as 1.0/0.0.
- Domain means the result is NAN (not a number), like 0.0/0.0.
- Overflow means the result is +INF (infinity) or -INF with complete loss of precision, such as assigning 1e200*1e200 to a double.

1.1.3.1.334 Floating point error: Stack fault

The floating-point stack has been overrun. This error may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function.

The program prints the error message and calls abort and _exit.

These floating-point errors can be avoided by masking the exception so that it doesn't occur, or by catching the exception with signal.

1.1.3.1.335 Floating point error: Partial loss of precision OR Floating point error: Underflow

These exceptions are masked by default, because underflows are converted to zero and losses of precision are ignored.

1.1.3.1.336 File name too long (E2197)

The file name given in an `#include` directive was too long for the compiler to process.

File names in DOS must be no more than 79 characters long.

1.1.3.1.337 Cannot evaluate function call (E2195)

The error message is issued if someone tries to explicitly construct an object or call a virtual function.

In integrated debugger expression evaluation, calls to certain functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.

1.1.3.1.338 Not a valid expression format type (E2198)

Invalid format specifier following expression in the debug evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).

1.1.3.1.339 Functions may not be part of a struct or union (E2200)

This C struct or union field was declared to be of type function rather than pointer to function.

Functions as fields are allowed only in C++.

1.1.3.1.340 Too much global data defined in file (E2201)

The sum of the global data declarations exceeds 64K bytes. This includes any data stored in the DGROUP (all global variables, literal strings, and static locals).

Solutions

Check the declarations for any array that might be too large. You can also remove variables from the DGROUP.

Here's how:

- Declare the variables as automatic. This uses stack space.
- Dynamically allocate memory from the heap using calloc, malloc, or farmalloc for the variables. This requires the use of pointers.

Literal strings are also put in the DGROUP. Get the file farstr.zip from our BBS to extract literal strings into their own segment.

1.1.3.1.341 Goto into an exception handler is not allowed (E2202)

It is not legal to jump into a try block, or an exception handler that is attached to a try block.

1.1.3.1.342 Goto bypasses initialization of a local variable (E2203)

In C++, it is illegal to bypass the initialization of a local variable.

This error indicates a goto statement that can transfer control past this local variable.

1.1.3.1.343 Group overflowed maximum size: 'name' (E2204)

The total size of the segments in a group (for example, DGROUP) exceeded 64K.

1.1.3.1.344 Illegal type type in __automated section (E2205)

Only certain types are allowed in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
long__fastcall fooLong(long);// Error: long illegal
};
```

1.1.3.1.345 Illegal character 'character' (0x'value') (E2206)

The compiler encountered some invalid character in the input file.

The hexadecimal value of the offending character is printed.

This can also be caused by extra parameters passed to a function macro.

1.1.3.1.346 Implicit conversion of 'type1' to 'type2' not allowed (E2207)

When a member function of a class is called using a pointer to a derived class, the pointer value must be implicitly converted to

point to the appropriate base class.

In this case, such an implicit conversion is illegal.

1.1.3.1.347 Cannot access an inactive scope (E2208)

You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

1.1.3.1.348 Unable to open include file 'filename' (E2209)

The compiler could not find the named file.

Possible Causes

- The named file does not exist.
- An `#include` file included itself.
- You do not have FILES set in CONFIG.SYS on your root directory.

Solutions

- Verify that the named file exists.
- Set FILES = 20 in CONFIG.SYS.

1.1.3.1.349 Reference member 'member' is not initialized (E2210)

References must always be initialized, in the constructor for the class.

A class member of reference type must have an initializer provided in all constructors for that class.

This means you can't depend on the compiler to generate constructors for such a class, because it has no way of knowing how to initialize the references.

1.1.3.1.350 Function defined inline after use as extern (E2212)

Functions can't become inline after they have already been used.

Either move the inline definition forward in the file or delete it entirely.

The compiler encountered something like:

```
myex();
twoex() { myex(); }
inline myex() { return 2; } // error
```

and already used the function as an extern before it saw that it was specified as inline. This would be correct:

```
myex();
inline myex() { return 2; }
twoex() { myex(); }
```

or better:

```
inline myex();
inline myex() { return 2; }
twoex() { myex(); }
```

1.1.3.1.351 Inline assembly not allowed in inline and template functions (E2211)

The compiler can't handle inline assembly statements in a C++ inline or template function.

You could eliminate the inline assembly code or, in the case of an inline function, make this a macro, and remove the inline storage class.

1.1.3.1.352 Internal code generator error (F1001)

An error has occurred in the internal logic of the code generator. Contact CodeGear technical support.

1.1.3.1.353 Invalid template declaration (E2413)

After the declarator of a template member, either a semicolon, an initialization, or a body was expected, but some other, illegal token was found. This message appears when a template member is declared outside of the template, but the syntax was wrong.

1.1.3.1.354 Invalid use of namespace 'identifier' (E2070)

A namespace identifier was used in an illegal way, for example, in an expression.

1.1.3.1.355 Cannot have a 'non-inline function/static data' in a local class (E2214)

All members of classes declared local to a function must be entirely defined in the class definition.

This means that local classes cannot contain any static data members, and all of their member functions must have bodies defined within the class definition.

1.1.3.1.356 Linkage specification not allowed (E2215)

Linkage specifications such as extern "C" are only allowed at the file level.

Move this function declaration out to the file level.

1.1.3.1.357 Unable to create turboc.$ln (E2216)

The compiler cannot create the temporary file TURBOC.$LN because it cannot access the disk or the disk is full.

1.1.3.1.358 Local data exceeds segment size limit (E2217)

The local variables in the current function take up more than 64K.

1.1.3.1.359 Templates can only be declared at namespace or class scope (E2218)

Templates cannot be declared inside classes or functions. They are only allowed in the global scope, or file level.

For example:

```
void func()
{
  template <class T> myClass {  // Error
    T i;
  };
}
```

1.1.3.1.360 Wrong number of arguments in call of macro 'macro' (E2219)

Your source file called the named macro with an incorrect number of arguments.

1.1.3.1.361 Invalid macro argument separator (E2220)

In a macro definition, arguments must be separated by commas.

The compiler encountered some other character after an argument name.

This is correct:

```
#define tri_add(a, b, c)  ((a) + (b) + (c))
```

This is incorrect:

```
#define tri_add(a  b. c)  ((a) + (b) + (c))
```

1.1.3.1.362 Macro argument syntax error (E2221)

An argument in a macro definition must be an identifier.

The compiler encountered some non-identifier character where an argument was expected.

1.1.3.1.363 Macro expansion too long (E2222)

A macro can't expand to more than 4,096 characters.

1.1.3.1.364 Too many decimal points (E2223)

The compiler encountered a floating-point constant with more than one decimal point.

1.1.3.1.365 Too many exponents (E2224)

The compiler encountered more than one exponent in a floating-point constant.

1.1.3.1.366 Too many initializers (E2225)

The compiler encountered more initializers than were allowed by the declaration being initialized.

1.1.3.1.367 Extra parameter in call (E2226)

A call to a function, via a pointer defined with a prototype, had too many arguments.

1.1.3.1.368 Extra parameter in call to function (E2227)

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

1.1.3.1.369 Too many error or warning messages (E2228)

There were more errors or warnings than allowed.

1.1.3.1.370 Cannot initialize a class member here (E2233)

Individual members of structs, unions, and C++ classes can't have initializers.

A struct or union can be initialized as a whole using initializers inside braces.

A C++ class can only be initialized by the use of a constructor.

1.1.3.1.371 Constant/Reference member 'member' in class without constructors (E2232)

A class that contains constant or reference members (or both) must have at least one user-defined constructor.

Otherwise, there would be no way to ever initialize such members.

1.1.3.1.372 Member 'member' has the same name as its class (E2229)

A static data member, enumerator, member of an anonymous union, or nested type cannot have the same name as its class.

Only a member function or a non-static member can have a name that is identical to its class.

1.1.3.1.373 Memory reference expected (E2234)

The built-in assembler requires a memory reference.

You probably forgot to put square brackets around an index register operand.

1.1.3.1.374 Member 'member' cannot be used without an object (E2231)

This means that you have written class::member where 'member' is an ordinary (non-static) member, and there is no class to associate with that member.

For example, it is legal to write this:

```
obj.class::member
```

but not to write this:

```
class::member
```

1.1.3.1.375 Member function must be called or its address taken (E2235)

A reference to a member function must be called, or its address must be taken with & operator.

In this case, a member function has been used in an illegal context.

For example:

```
class A
{
   typedef int (A::* infptr)(void);
public:
   A();
   int myex(void);
   int three;
} a;
A::A()
{
   infptr one = myex;       //illegal - call myex or take address?
   infptr two = &A::myex;   //correct
   three = (a.*one)() + (a.*two)();
}
```

1.1.3.1.376 DPMI programs must use the large memory model (O2237)

The compiler no longer issues this error.

1.1.3.1.377 Multiple declaration for 'identifier' (E2238)

This identifier was improperly declared more than once.

This might be caused by conflicting declarations such as:

- int a; double a;
- a function declared two different ways, or
- a label repeated in the same function, or
- some declaration repeated other than an extern function or a simple variable

This can also happen by inadvertently including the same header file twice. For example, given:

```
//a.h
struct A { int a; };
//b.h
#include "a.h"
//myprog.cpp
#include "a.h"
#include "b.h"
```

myprog.cpp will get two declarations for the struct A. To protect against this, one would write the a.h header file as:

```
//a.h
#ifndef __A_H
#define __A_H
struct A { int a; };
#endif
```

This will allow one to safely include a.h several times in the same source code file.

1.1.3.1.378 'identifier' must be a member function (E2239)

Most C++ operator functions can be members of classes or ordinary non-member functions, but these are required to be members of classes:

- operator =
- operator ->
- operator ( )

- type conversions

This operator function is not a member function but should be.

1.1.3.1.379 Namespace name expected (E2282)

The name of a namespace symbol was expected.

1.1.3.1.380 Namespace member 'identifier' declared outside its namespace (E2334)

Namespace members must be declared inside their namespace. You can only use explicit qualification to define a namespace member (for example, to give a body for a function declared in a namespace). The declaration itself must be inside the namespace.

1.1.3.1.381 Conversion of near pointer not allowed (E2240)

A near pointer cannot be converted to a far pointer in the expression evaluation box when a program is not currently running. This is because the conversion needs the current value of DS in the user program, which doesn't exist.

1.1.3.1.382 Specifier requires Delphi style class type (E2242)

The stored, default, and nodefault storage specifiers are only allowed within property declarations of Delphi style class types.

Example

```
struct regclass
{
     int __property ip1 = { stored  = false }; // Error
     int __property ip2 = { default = 42    }; // Error
     int __property ip3 = { nodefault       }; // Error
};
struct __declspec(delphiclass) clxclass
{
     int __property ip1 = { stored  = false }; // OK
     int __property ip2 = { default = 42    }; // OK
     int __property ip3 = { nodefault       }; // OK
};
```

1.1.3.1.383 Array allocated using 'new' may not have an initializer (E2243)

When initializing a vector (array) of classes, you must use the constructor that has no arguments.

This is called the default constructor, which means that you can't supply constructor arguments when initializing such a vector.

1.1.3.1.384 'new' and 'delete' not supported (E2244)

The integrated debugger does not support the evaluation of the new and delete operators.

1.1.3.1.385 Cannot allocate a reference (E2245)

You have attempted to create a reference using the new operator.

This is illegal, because references are not objects and can't be created through new.

1.1.3.1.386 'member' is not accessible (E2247)

You are trying to reference C++ class member 'member,' but it is private or protected and can't be referenced from this function.

This sometimes happens when you attempt to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function.

The check for overload resolution is always made before checking for accessibility.

If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function.

Virtual base class constructors must be accessible within the scope of the most derived class. This is because C++ always constructs virtual base classes first, no matter how far down the hierarchy they are. For example:

```
class A {
public:
   A();
};
class B : private virtual A {};
class C : private B {
public:
C();
};
C::C() {} // error, A::A() is not accessible
```

Since A is private to B, which is private to C, it makes A's constructor not accessible to C. However, the constructor for C must be able to call the constructors for its virtual base class, A. If B inherits A publicly, the above example would compile.

1.1.3.1.387 Cannot find default constructor to initialize array element of type 'class' (E2248)

When declaring an array of a class that has constructors, you must either explicitly initialize every element of the array, or the class must have a default constructor.

The compiler will define a default constructor for a class unless you have defined any constructors for the class.

1.1.3.1.388 Inline assembly not allowed (E2309)

Your source file contains inline assembly language statements and you are compiling it from within the integrated environment.

You must use the BCC command to compile this source file from the DOS command line.

1.1.3.1.389 No base class to initialize (E2250)

This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes.

Check your declarations.

1.1.3.1.390 Cannot find default constructor to initialize base class 'class' (E2251)

Whenever a C++ derived class 'class2' is constructed, each base class 'class1' must first be constructed.

If the constructor for 'class2' does not specify a constructor for 'class1' (as part of 'class2's' header), there must be a constructor class1::class1() for the base class.

This constructor without parameters is called the default constructor.

The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class1'.

In that case, the compiler will not supply the default constructor automatically--you must supply one.

```
class Base {
public:
   Base(int) {}
};
class Derived = public Base {
   Derived():Base(1) {}
}
// must explicitly call the Base constructor, or provide a
// default constructor in Base.
```

Class members with constructors must be initialized in the class' initializer list, for example:

```
class A {
public
   A( int );
};
class B {
public:
   A a;
   B() : a( 3 ) {}; //ok
};
```

1.1.3.1.391 'catch' expected (E2252)

In a C++ program, a 'try' block must be followed by at least one 'catch' block.

1.1.3.1.392 Calling convention must be attributed to the function type, not the closure (E2253)

The calling convention is in the wrong place in the closure declaration. For example,

```
int __fastcall (__closure * x)()
```

will compile, but

```
int (__fastcall __closure * x)()
```

will not.

1.1.3.1.393 : expected after private/protected/private (E2254)

When used to begin a private, protected, or public section of a C++ class, the reserved words "private," "protected," and "public" must be followed by a colon.

1.1.3.1.394 Use :: to take the address of a member function (E2255)

If f is a member function of class c, you take its address with the syntax

```
&c::f
```

Note the use of the class type name (not the name of an object) and the :: separating the class name from the function name.

(Member function pointers are not true pointer types, and do not refer to any particular instance of a class.)

1.1.3.1.395 No : following the ? (E2256)

The question mark (?) and colon (:) operators do not match in this expression.

The colon might have been omitted, or parentheses might be improperly nested or missing.

1.1.3.1.396 , expected (E2257)

A comma was expected in a list of declarations, initializations, or parameters.

This problem is often caused by a missing syntax element earlier in the file or one of its included headers.

1.1.3.1.397 Declaration was expected (E2258)

A declaration was expected here but not found.

This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.

1.1.3.1.398 Default value missing (E2259)

When a C++ function declares a parameter with a default value, all of the following parameters must also have default values.

In this declaration, a parameter with a default value was followed by a parameter without a default value.

1.1.3.1.399 Default value missing following parameter 'parameter' (E2260)

All parameters following the first parameter with a default value must also have defaults specified.

1.1.3.1.400 Use of dispid with a property requires a getter or setter (E2261)

This property needs either a getter or a setter.

1.1.3.1.401 Exception handling not enabled (E2263)

A 'try' block was found with the exception handling disabled.

1.1.3.1.402 '__except' or '__finally' expected following '__try' (E2262)

In C, a '__try block' must be followed by a '__except' or '__finally' handler block.

1.1.3.1.403 Expression expected (E2264)

An expression was expected here, but the current symbol can't begin an expression.

This message might occur where the controlling expression of an if or while clause is expected or where a variable is being initialized.

This message is often due to a symbol that is missing or has been added.

1.1.3.1.404 No file names given (E2266)

The command line contained no file names. You must specify a source file name.

1.1.3.1.405 No file name ending (E2265)

The file name in an `#include` statement was missing the correct closing quote or angle bracket.

1.1.3.1.406 Goto statement missing label (E2271)

The `goto` keyword must be followed by an identifier.

1.1.3.1.407 > expected (E2270)

A new-style cast (for example, dynamic_cast) was found with a missing closing ">".

1.1.3.1.408 Identifier expected (E2272)

An identifier was expected here, but not found.

In C, an identifier is expected in the following situations:

- in a list of parameters in an old-style function header
- after the reserved words struct or union when the braces are not present, and
- as the name of a member in a structure or union (except for bit fields of width 0).

In C++, an identifier is also expected in these situations:

- in a list of base classes from which another class is derived, following a double colon (::), and
- after the reserved word "operator" when no operator symbol is present.

1.1.3.1.409 'main' cannot be declared as static or inline (E2273)

You cannot make main static or inline. For example, you cannot use static int main() or inline int main().

1.1.3.1.410 Opening brace expected (E2275)

A left brace was expected at the start of a block or initialization.

1.1.3.1.411 (expected (E2276)

A left parenthesis was expected before a parameter list.

1.1.3.1.412 < expected (E2274)

The keyword `template` was not followed by <.

Every template declaration must include the template formal parameters enclosed within < >, immediately following the template keyword.

1.1.3.1.413 Lvalue required (E2277)

The left side of an assignment operator must be an addressable expression.

Addressable expressions include the following:

- numeric or pointer variables
- structure field references or indirection through a pointer
- a subscripted array element

1.1.3.1.414 Multiple base classes not supported for Delphi classes (E2278)

Delphi style classes cannot have multiple base classes.

Example

```
struct__declspec(delphiclass) base1  {};
struct__declspec(delphiclass) base2  {};
structderived : base1, base2         {}; // Error
```

1.1.3.1.415 Member identifier expected (E2280)

The name of a structure or C++ class member was expected here, but not found. The right side of a dot (.) or arrow (->) operator must be the name of a member in the structure or class on the left of the operator.

1.1.3.1.416 Cannot find default constructor to initialize member 'identifier' (E2279)

When the following occurs

1.A C++ class 'class1' contains a member of class 'class2,'

and

2.You want to construct an object of type 'class1' (but not from another object of type 'class1'). There must be a constructor class2::class2() so that the member can be constructed.

This constructor without parameters is called the default constructor.

The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class2'.

In that case, the compiler will not supply the default constructor automatically (you must supply one.

1.1.3.1.417 Identifier1 requires definition of Identifier2 as a pointer type (E2281)

To use Identifier1, there needs to be a definition for Identifier2, which is a type.

Example where __classid is the first identifier and TClass, which can be found in clx.h, is the second one:

```
// #include <clx/clx.h> missing
struct __declspec(delphiclass)bar
{
     virtual int barbara(void);
};
void *foo(void)
```

```
{
     return classid(bar);    // Error
}
```

1.1.3.1.418 Only member functions may be 'const' or 'volatile' (E2310)

Something other than a class member function has been declared `const` or `volatile`.

1.1.3.1.419 Non-virtual function 'function' declared pure (E2311)

Only virtual functions can be declared pure, because derived classes must be able to override them.

1.1.3.1.420 Use . or -> to call 'function' (E2283)

You attempted to call a member function without providing an object. This is required to call a member function.

```
class X {
   member func() {}
};
X x;
X*xp = new X;
X.memberfunc();
Xp-> memberfunc();
```

1.1.3.1.421 Use . or -> to call 'member', or & to take its address (E2284)

A reference to a non-static class member without an object was encountered.

Such a member can't be used without an object, or its address must be taken with the & operator.

1.1.3.1.422 Could not find a match for 'argument(s)' (E2285)

No C++ function could be found with parameters matching the supplied arguments. Check parameters passed to function or overload function for parameters that are being passed.

1.1.3.1.423 Overloaded function resolution not supported (E2286)

In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.

1.1.3.1.424 Parameter 'number' missing name (E2287)

In a function definition header, this parameter consisted only of a type specifier 'number' with no parameter name.

This is not legal in C.

(It is allowed in C++, but there's no way to refer to the parameter in the function.)

1.1.3.1.425 Pointer to structure required on left side of -> or ->* (E2288)

Nothing but a pointer is allowed on the left side of the arrow (->) in C or C++.

In C++ a -> operator is allowed.

1.1.3.1.426 __published or __automated sections only supported for Delphi classes (E2289)

The compiler needs to generate a special kind of vtable for classes containing __published and __automated sections. Therefore, these sections are only supported for Delphi style classes.

Example

```
structregclass
{
int mem;
__published:// Error: no Delphi style class
int __property ip = { read = mem, write = mem };
};
struct__declspec(delphiclass) clxclass
{
int mem;
__published:// OK
int __property ip = { read = mem, write = mem };
};
```

1.1.3.1.427 'code' missing] (E2290)

This error is generated if any of the following occur:

- Your source file declared an array in which the array bounds were not terminated by a right bracket.
- The array specifier in an operator is missing a right bracket.
- The operator [ ] was declared as operator [.
- A right bracket is missing from a subscripting expression.

Add the bracket or fix the declaration.

Check for a missing or extra operator or mismatched parentheses.

1.1.3.1.428 brace expected (E2291)

A right brace was expected at the end of a block or initialization.

1.1.3.1.429 Function should return a value (E2292)

Your source file declared the current function to return some type other than `int` or `void`, but the compiler encountered a return with no value. This usually indicates some sort of error.

Functions declared as returning `int` are exempt because older versions of C did not support `void` function return types.

1.1.3.1.430) expected (E2293)

A right parenthesis was expected at the end of a parameter list.

1.1.3.1.431 Structure required on left side of . or .* (E2294)

The left side of a dot (.) operator (or C++ dot-star operator, .*) must evaluate to a structure type. In this case it did not.

This error can occur when you create an instance of a class using empty parentheses, and then try to access a member of that

'object'.

1.1.3.1.432 'constructor' is not an unambiguous base class of 'class' (E2312)

A C++ class constructor is trying to call a base class constructor 'constructor.'

This error can also occur if you try to change the access rights of 'class::constructor.'

Check your declarations.

1.1.3.1.433 Constant expression required (E2313)

Arrays must be declared with constant size.

This error is commonly caused by misspelling a `#define` constant.

1.1.3.1.434 Templates not supported (E2296)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

1.1.3.1.435 Call of nonfunction (E2314)

The name being called is not declared as a function.

This is commonly caused by incorrectly declaring the function or misspelling the function name.

1.1.3.1.436 Declaration does not specify a tag or an identifier (E2321)

This declaration doesn't declare anything.

This may be a struct or union without a tag or a variable in the declaration. C++ requires that something be declared.

For example:

```
struct
{
int a
};
//no tag or identifier
```

1.1.3.1.437 'this' can only be used within a member function (E2297)

In C++, "this" is a reserved word that can be used only within class member functions.

1.1.3.1.438 'identifier' is not a member of 'struct' (E2316)

You are trying to reference 'identifier' as a member of 'struct', but it is not a member.

Check your declarations.

1.1.3.1.439 'Member' is not a member of 'class', because the type is not yet defined (E2315)

The member is being referenced while the class has not been fully defined yet. This can happen if you forward declare class X, declare a pointer variable to X, and reference a member through that pointer; for example:

```
class X;
X * oneX;
int test() { return oneX->i; }
```

1.1.3.1.440 Cannot generate 'function' from template function 'template' (E2298)

A call to a template function was found, but a matching template function cannot be generated from the function template.

1.1.3.1.441 Cannot use templates in closure arguments -- use a typedef (E2301)

When declaring a closure type, the arguments passed to that closure must be of a simple type. Templates are not accepted. To pass a reference to an object of template type to a closure, you must declare a typedef, which counts as a simple type name.

Example

```
typedef my_class<int> mci;
typedef void (__closure * func) (const mci& object);
```

1.1.3.1.442 'identifier' is not a parameter (E2317)

In the parameter declaration section of an old-style function definition, 'identifier' is declared but not listed as a parameter. Either remove the declaration or add 'identifier' as a parameter.

1.1.3.1.443 'type' is not a polymorphic class type (E2318)

This error is generated if the -RT compiler option (for runtime type information) is disabled and either

dynamic_cast was used with a pointer to a class

or

you tried to delete a pointer to an object of a class that has a virtual destructor

1.1.3.1.444 'identifier' is not a public base class of 'classtype' (E2319)

The right operand of a .*, ->*, or ::operator was not a pointer to a member of a class that is either identical to (or an unambiguous accessible base class of) the left operand's class type.

1.1.3.1.445 Expression of scalar type expected (E2320)

The !, ++, and -- operators require an expression of scalar type.

Only these types are allowed:

- char

- short
- int
- long
- enum
- float
- double
- long double
- pointer

1.1.3.1.446 No type information (E2302)

The integrated debugger has no type information for this variable. Ensure that you've compiled the module with debug information. If it has, the module may have been compiled by another compiler or assembler.

1.1.3.1.447 Type name expected (E2303)

One of these errors has occurred:

- In declaring a file-level variable or a struct field, neither a type name nor a storage class was given.
- In declaring a typedef, no type for the name was supplied.
- In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class).
- In supplying a C++ base class name, the name was not the name of a class.

1.1.3.1.448 'Constant/Reference' variable 'variable' must be initialized (E2304)

This C++ object is declared constant or as a reference, but is not initialized.

It must be initialized at the point of declaration.

1.1.3.1.449 Cannot find 'class::class' ('class'&) to copy a vector OR Cannot find 'class'::operator=('class'&) to copy a vector (E2305)

When a C++ class 'class1' contains a vector (array) of class 'class2', and you want to construct an object of type 'class1' from another object of type 'class 1', you must use this constructor:

```
class2::class2(class2&)
```

so that the elements of the vector can be constructed.

The constructor, called a copy constructor, takes just one parameter (which is a reference to its class).

Usually, the compiler supplies a copy constructor automatically.

However, if you have defined a constructor for class 'class2' that has a parameter of type 'class2&' and has additional parameters with default values, the copy constructor can't exist and can't be created by the compiler.

This is because these two can't be distinguished:

```
class2::class2(class2&)
class2::class2(class2&, int = 1)
```

You must redefine this constructor so that not all parameters have default values.

You can then define a reference constructor or let the compiler create one.

Cannot find class::operator= ...

When a C++ class 'class1' contains a vector (array) of class 'class2', and you want to copy a class of type 'class1', you must use this assignment operator:

```
class2::class2(class2&)
```

so that the elements of the vector can be copied.

Usually, the compiler automatically supplies this operator.

However, if you have defined an operator= for class 'class2' that does not take a parameter of type 'class2&,' the compiler will not supply it automatically--you must supply one.

1.1.3.1.450 Virtual base classes not supported for Delphi classes (E2306)

Delphi style classes cannot be derived virtually, not even from other Delphi style classes.

Example

```
struct __declspec(delphiclass) base  {};
struct derived : virtual base        {};    // Error
```

1.1.3.1.451 Type 'type' is not a defined class with virtual functions (E2307)

A dynamic_cast was used with a pointer to a class type that is either undefined, or doesn't have any virtual member functions.

1.1.3.1.452 do statement must have while (E2308)

Your source file contained a do statement that was missing the closing while keyword.

1.1.3.1.453 Incorrect number format (E2322)

The compiler encountered a decimal point in a hexadecimal number.

1.1.3.1.454 Illegal number suffix (E2323)

A numeric literal is followed by a suffix that is not recognized by the compiler.

Example

```
int i = 1234i15;   // Error: no i15 suffix
int j = 1234i16;   // OK
```

1.1.3.1.455 Numeric constant too large (E2324)

String and character escape sequences larger than hexadecimal or octal 77 can't be generated.

Two-byte character constants can be specified by using a second backslash. For example,

```
\\
```

represents a two-byte constant.

A numeric literal following an escape sequence should be broken up like this:

```
printf("\x0A" "12345");
```

This prints a carriage return followed by 12345.

1.1.3.1.456 Illegal octal digit (E2325)

The compiler found an octal constant containing a non-octal digit (8 or 9).

1.1.3.1.457 Use __declspec(spec1[, spec2]) to combine multiple __declspecs (E2326)

When you want to use several __declspec modifiers, the compiler will complain if you don't combine them into one __declspec. For example:

```
int __declspec(__import) __declspec(delphiclass) X
```

will give an error. Use the following instead:

```
int __declspec(__import, delphiclass) X
```

1.1.3.1.458 Invalid combination of opcode and operands (E2329)

The built-in assembler does not accept this combination of operands.

Possible causes

- There are too many or too few operands for this assembler opcode.
- The number of operands is correct, but their types or order do not match the opcode.

1.1.3.1.459 Operators may not have default argument values (E2327)

It is illegal for overloaded operators to have default argument values.

1.1.3.1.460 Classes with properties cannot be copied by value (E2328)

This error occurs if you attempt to use the default assignment operator. For example, the following code generates this error given two labels on a form:

```
*Label1->Font = *Label2->Font;
```

1.1.3.1.461 Operator must be declared as function (E2330)

An overloaded operator was declared with something other than function type.

For example:

```
class A
{
   A& operator +;   ..note missing parenthesis
};
```

In the example, the function operator '()' is missing, so the operator does not have function type and generates this error.

1.1.3.1.462 Number of allowable option contexts exceeded (E2331)

You have interspersed too many source-code option changes (using #pragma option) between template declarations. For example:

```
#pragma option -x
template<class T> class foo1 { };
#pragma option -a3
template<class T> class foo2 { };
#pragma option -b
template<class T> class foo3 { };
#pragma option -k-
```

You need to break your source code into smaller files.

1.1.3.1.463 Variable 'variable' has been optimized and is not available (E2332)

You have tried to inspect, watch, or otherwise access a variable which the optimizer removed.

This variable is never assigned a value and has no stack location.

1.1.3.1.464 Class member 'member' declared outside its class (E2333)

C++ class member functions can be declared only inside the class declaration.

Unlike nonmember functions, they can't be declared multiple times or at other locations.

1.1.3.1.465 Overloaded 'function name' ambiguous in this context (E2335)

The only time an overloaded function name can be used or assigned without actually calling the function is when a variable or parameter of the correct function pointer type is initialized or assigned the address of the overload function.

In this case, an overloaded function name has been used in some other context, for example, the following code will generate this error:

```
class A{
   A(){myex;}           //calling the function
   void myex(int) {}    //or taking its address?
   void myex(float){}
};
```

1.1.3.1.466 Cannot overload 'function' (E2476)

You cannot overload the specified function. This error is displayed if you tried to declare a function with the same name as another function, but the redeclaration is not legal. For example, if both functions have the 'extern "C"' linkage type, only one 'extern "C"' function can have a given name.

1.1.3.1.467 Cannot overload 'main' (E2339)

You cannot overload main.

1.1.3.1.468 Pointer to overloaded function 'function' doesn't match 'type' (E2336)

A variable or parameter is assigned (or initialized with) the address of an overloaded function.

However, the type of the variable or parameter doesn't match any of the overloaded functions with the specified name.

1.1.3.1.469 Only one of a set of overloaded functions can be "C" (E2337)

C++ functions are by default overloaded, and the compiler assigns a new name to each function.

If you wish to override the compiler's assigning a new name by declaring the function `extern` "C", you can do this for only one of a set of functions with the same name.

(Otherwise the linker would find more than one global function with the same name.)

1.1.3.1.470 Overlays only supported in medium, large, and huge memory models (E2338)

The compiler no longer issues this error.

1.1.3.1.471 Type mismatch in parameter 'number' (E2340)

The function called, via a function pointer, was declared with a prototype.

However, the given parameter number (counting left to right from 1) could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.472 Type mismatch in parameter 'number' in call to 'function' (E2341)

Your source file declared the named function with a prototype, and the given parameter number (counting left to right from 1) could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'", but the mismatch might be due to many other reasons.

1.1.3.1.473 Type mismatch in parameter 'parameter' (E2342)

Your source file declared the function called via a function pointer with a prototype.

However, the named parameter could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.474 Type mismatch in parameter 'parameter' in call to 'function' (E2343)

Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.475 Access can only be changed to public or protected (E2345)

A C++ derived class can modify the access rights of a base class member, but only to public or protected.

A base class member can't be made private.

1.1.3.1.476 'x' access specifier of property 'property' must be a member function (E2346)

Only member functions or data members are allowed in access specifications of properties.

Example

```
int GlobalGetter(void)
{
     return 0;
}
struct pbase
{
     intMemberGetter(void) {return 1;}
     int __property ip1 = { read = GlobalGetter }; // Error
     int __property ip2 = { read = MemberGetter }; // OK
};
```

1.1.3.1.477 Parameter mismatch in access specifier 'specifier' of property 'property' (E2347)

The parameters of the member function used to access a property don't match the expected parameters.

Example

```
structpbase
{
       voidSetter1(void){}
       voidSetter2(int){}
     int __property ip1 = { write = Setter1 }; // Error
     int __property ip2 = { write = Setter2 }; // OK
};
```

1.1.3.1.478 Storage specifier not allowed for array properties (E2348)

Array properties cannot have a storage specification.

Example

```
struct pbase
{
    int __property ap[char *] =
                           { stored = false }; // Error
};
```

1.1.3.1.479 Nonportable pointer conversion (E2349)

An implicit conversion between a pointer and an integral type is required, but the types are not the same size. You must use an explicit cast.

This conversion might not make any sense, so be sure this is what you want to do.

1.1.3.1.480 Cannot define a pointer or reference to a reference (E2350)

It is illegal to have a pointer to a reference or a reference to a reference.

1.1.3.1.481 Static data members not allowed in __published or __automated sections (E2351)

Only nonstatic data members and member functions are allowed in __published or __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__published:
static intstaticDataMember;// Error
};
```

1.1.3.1.482 Cannot create instance of abstract class 'class' (E2352)

Abstract classes (those with pure virtual functions) can't be used directly, only derived from.

When you derive an abstract base class, with the intention to instantiate instances of this derived class, you must override each of the pure virtual functions of the base class exactly as they are declared.

For example:

```
class A {
public:
   virtual myex( int ) = 0;
   virtual twoex( const int ) const = 0;
};
class B : public A {
public:
   myex( int );
   twoex( const int );
};
B b;   // error
```

The error occurs because we have not overridden the virtual function in which twoex can act on const objects of the class. We have created a new one which acts on non-const objects. This would compile:

```
class A {
public:
   virtual myex( int ) = 0;
   virtual twoex( const int ) const = 0;
};
class B : public A {
public:
   myex( int );
   twoex( const int ) const;
};
B b;   // ok
```

1.1.3.1.483 Class 'classname' is abstract because of 'member = 0' (E2353)

This message is issued immediately after the "Cannot create instance of abstract class 'classname' error message and is intended to make it easier to figure out why a particular class is considered abstract by the compiler.

For example, consider the following example of an illegal attempt to instantiate an abstract class:

```
struct VB
```

```
{
virtualvoid  f() = 0;
virtualvoid  g() = 0;
virtualvoid  h() = 0;
};
struct D1 : virtual VB
{
void   f();
};
struct D2 : virtual VB
{
void   h();
};
struct DD : D1, D2
{
}
v;   // error 'DD' is an abstract class
```

The above code will cause the following two error messages:

```
Error TEST.CPP 21: Cannot create instance of abstract class 'DD'
Error TEST.CPP 21: Class 'DD' is abstract because of 'VB::g() = 0'
```

1.1.3.1.484 Two operands must evaluate to the same type (E2354)

The types of the expressions on both sides of the colon in the conditional expression operator (?:) must be the same, except for the usual conversions.

These are some examples of usual conversions

- `char` to `int`
- `float` to `double`
- void* to a particular pointer

In this expression, the two sides evaluate to different types that are not automatically converted.

This might be an error or you might merely need to cast one side to the type of the other.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.485 Recursive template function: 'x' instantiated 'y' (E2355)

The compiler has detected a recursive template function instance. For example:

```
template<class T> void f(T x)
{
   f((T*)0);   // recursive template function!
}
void main()
{
   f(0);
}
```

The compiler issue one message for each nesting of the recursive instantiation, so it is usually obvious where the recursion has occurred. To fix a recursive template, either change the dependencies or provide a specialized version that will stop the recursion. For example, adding the following function definition to the above program will remove the endless recursion:

```
void f(int **)
{
}
```

1.1.3.1.486 Type mismatch in redeclaration of 'identifier' (E2356)

Your source file redeclared a variable with a different type than was originally declared for the variable.

Possible Causes

This can occur if a function is called and subsequently declared to return something other than an integer.

Solutions

If this has happened, you must declare the function before the first call to it.

1.1.3.1.487 Reference initialized with 'type1', needs lvalue of type 'type2' (E2357)

A reference variable that is not declared constant must be initialized with an lvalue of the appropriate type.

In this case, the initializer either wasn't an lvalue, or its type didn't match the reference being initialized.

1.1.3.1.488 Reference member 'member' needs a temporary for initialization (E2358)

You provided an initial value for a reference type that was not an lvalue of the referenced type.

This requires the compiler to create a temporary for the initialization.

Because there is no obvious place to store this temporary, the initialization is illegal.

1.1.3.1.489 Reference member 'member' initialized with a non-reference parameter (E2359)

An attempt has been made to bind a reference member to a constructor parameter. Since the parameter will cease to exist the moment the constructor returns to its caller, this will not work correctly.

1.1.3.1.490 Invalid register combination (e.g. [BP+BX]) (E2360)

The built-in assembler detected an illegal combination of registers in an instruction.

These are valid index register combinations:

- [BX]
- [BP]
- [SI]
- [DI]
- [BX+SI]
- [BX+DI]
- [BP+SI]
- [BP+DI]

Other index register combinations are not allowed.

1.1.3.1.491 'specifier' has already been included (E2361)

This type specifier occurs more than once in this declaration.

Delete or change one of the occurrences.

1.1.3.1.492 Repeat count needs an lvalue (E2362)

The expression before the comma (,) in the Watch or Evaluate window must be an accessible region of storage. For example, expressions like this one are not valid:

```
i++,10d
x = y, 10m
```

1.1.3.1.493 Attempting to return a reference to local variable 'identifier' (E2363)

This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable.

This is illegal, because the variable referred to disappears when the function exits.

You can return a reference to any static or global variable, or you can change the function to return a value instead.

1.1.3.1.494 Attempting to return a reference to a local object (E2364)

You attempted to return a reference to a temporary object in a function that returns a reference type. This may be the result of a constructor or a function call.

This object will disappear when the function returns, making the reference illegal.

1.1.3.1.495 Member pointer required on right side of .* or ->* (E2365)

The right side of a C++ dot-star (.*) or an arrow star (->*) operator must be declared as a pointer to a member of the class specified by the left side of the operator.

In this case, the right side is not a member pointer.

1.1.3.1.496 Can't inherit non-RTTI class from RTTI base OR E2367 Can't inherit RTTI class from non-RTTI base (E2366)

When virtual functions are present, the RTTI attribute of all base classes must match that of the derived class.

1.1.3.1.497 RTTI not available for expression evaluation (E2368)

Expressions requiring RTTI are not supported by the expression evaluator in the integrated debugger. This error message is only issued by the expression evaluator (if you try to Inspect, Watch, or Evaluate), not by the compiler.

1.1.3.1.498 Cannot use the result of a property assignment as an rvalue' (E2369)

The result of a property assignment is an lvalue. This implies for instance that chained assignments of properties is not allowed;

for example, x = y = 5, where both x and y are properties. Certain embedded assignments of properties can also produce errors; for example, x != (y = z), where y is a property.

1.1.3.1.499 Simple type name expected (E2370)

To ensure interoperability between Delphi and C++, there are restrictions on the type names mentioned in the parameter lists of published closure types. The parameter types have to be simple type names with optional const modifier and pointer or reference notation.

So when declaring a closure type, the arguments passed to that closure must be of a simple type. For example, templates are not accepted. To pass a reference to an object of template type to a closure, you must declare a typedef, which counts as a simple type name.

Example

```
struct __declspec(delphiclass) foo
{
     typedef void __fastcall (__closure *foo1)(SomeTemplateType<int> *);
     typedef SomeTemplateType<int> SimpleTypeName;
     typedef void __fastcall (__closure *foo2)(SimpleTypeName *);
published:
     __property foo1 prop1;  // Error
     __property foo2 prop2;  // OK
};
```

1.1.3.1.500 sizeof may not be applied to a bit field (E2371)

sizeof returns the size of a data object in bytes, which does not apply to a bit field.

1.1.3.1.501 sizeof may not be applied to a function (E2372)

sizeof can be applied only to data objects, not functions.

You can request the size of a pointer to a function.

1.1.3.1.502 Specialization after first use of template (E2433)

An ANSI C++ rule requires that a specialization for a function template be declared before its first use. This error message is only issued when the ANSI conformance option (-A) is active.

1.1.3.1.503 Bit field cannot be static (E2373)

Only ordinary C++ class data members can be declared static, not bit fields.

1.1.3.1.504 Function 'function' cannot be static (E2374)

Only ordinary member functions and the operators new and delete can be declared static.

Constructors, destructors and other operators must not be static.

1.1.3.1.505 Stack overflow

This error is reported when you compile a function with the Test Stack Overflow option on, but there is not enough stack space to

allocate the function's local variables.

This error can also be caused by the following:

- infinite recursion, or
- an assembly language procedure that does not maintain the stack project
- a large array in a function

1.1.3.1.506 statement missing ((E2376)

In a do, for, if, switch, or while statement, the compiler found no left parenthesis after the while keyword or test expression.

1.1.3.1.507 statement missing) (E2377)

In a do, for, if, switch, or while statement, the compiler found no right parenthesis after the while keyword or test expression.

1.1.3.1.508 do-while or for statement missing ; (E2378)

In a do or for statement, the compiler found no semicolon after the right parenthesis.

1.1.3.1.509 Statement missing ; (E2379)

The compiler encountered an expression statement without a semicolon following it.

1.1.3.1.510 Unterminated string or character constant (E2380)

The compiler found no terminating quote after the beginning of a string or character constant.

1.1.3.1.511 Structure size too large (E2381)

Your source file declared a structure larger than 64K.

1.1.3.1.512 Side effects are not allowed (E2382)

Side effects such as assignments, ++, or -- are not allowed in the debugger watch window. A common error is to use x = y (not allowed) instead of x == y to test the equality of x and y.

1.1.3.1.513 Switch selection expression must be of integral type (E2383)

The selection expression in parentheses in a `switch` statement must evaluate to an integral type (`char, short, int, long, enum`).

You might be able to use an explicit cast to satisfy this requirement.

1.1.3.1.514 Templates and overloaded operators cannot have C linkage (E2132)

You tried to use a linkage specification with a template or overloaded operator. The most common cause for this error message

is having the declaration wrapped in an extern "C" linkage specification.

1.1.3.1.515 Cannot call near class member function with a pointer of type 'type' (E2384)

Also E2385 Cannot call near class member function 'function' with a pointer of type 'type'

Member functions of near classes can't be called via a member pointer.

This also applies to calls using pointers to members.

(Remember, classes are near by default in the tiny, small, and medium memory models.)

Either change the pointer to be near, or declare the class as far.

1.1.3.1.516 Type mismatch in parameter 'number' in template class name 'template' (E2390)

The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.517 Type mismatch in parameter 'parameter' in template class name 'template' (E2391)

The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

1.1.3.1.518 Too few arguments passed to template 'template' (E2394)

A template class name was missing actual values for some of its formal parameters.

1.1.3.1.519 Too many arguments passed to template 'template' (E2395)

A template class name specified too many actual values for its formal parameters.

1.1.3.1.520 Template argument must be a constant expression (E2396)

A non-type template class argument must be a constant expression of the appropriate type.

This includes constant integral expressions and addresses of objects or functions with external linkage or members.

1.1.3.1.521 Template function argument 'argument' not used in argument types (E2398)

The given argument was not used in the argument list of the function.

The argument list of a template function must use all of the template formal arguments; otherwise, there is no way to generate a template function instance based on actual argument types.

1.1.3.1.522 Invalid template argument list (E2401)

This error indicates that an illegal template argument list was found.

In a template declaration, the keyword `template` must be followed by a list of formal arguments enclosed within < and > delimiters.

1.1.3.1.523 Nontype template argument must be of scalar type (E2400)

A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.

1.1.3.1.524 Template functions may only have 'type-arguments' (E2415)

A function template was declared with a non-type argument.

This is not allowed with a template function, as there is no way to specify the value when calling it.

1.1.3.1.525 Error while instantiating template 'template' (E2419)

An error occurred during the instantiation of a particular template. This message always follows some other error message that indicates what actually went wrong. This message is displayed to help track down which template instantiation introduced the problem.

1.1.3.1.526 Template class nesting too deep: 'class' (E2424)

The compiler imposes a certain limit on the level of template class nesting. This limit is usually only exceeded through a recursive template class dependency.

When this nesting limit is exceeded, the compiler issues this error message for all of the nested template classes. This usually makes it easy to spot the recursion.

This error message is always followed by the fatal error "Out of memory".

1.1.3.1.527 'member' is not a valid template type member (E2425)

A member of a template with some actual arguments that depend on the formal arguments of an enclosing template was found not to be a member of the specified template in a particular instance.

1.1.3.1.528 Templates must be classes or functions (E2428)

The declaration in a template declaration must specify either a class type or a function.

1.1.3.1.529 'template' qualifier must name a template class or function instance' (E2432)

When defining a template class member, the actual arguments in the template class name used as the left operand for the :: operator must match the formal arguments of the template class.

1.1.3.1.530 Two consecutive dots (E2442)

Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), two consecutive dots cannot legally occur in a C program.

1.1.3.1.531 Base class 'class' is initialized more than once (E2443)

In a C++ class constructor, the list of initializations following the constructor header includes base class 'class' more than once.

1.1.3.1.532 Member 'member' is initialized more than once (E2444)

In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.

1.1.3.1.533 Variable 'identifier' is initialized more than once (E2445)

This variable has more than one initialization. It is legal to declare a file level variable more than once, but it can have only one initialization (even if two are the same).

1.1.3.1.534 Function definition cannot be a typedef'ed declaration (E2446)

In ANSI C, a function body cannot be defined using a typedef with a function Type.

Redefine the function body.

1.1.3.1.535 'identifier' must be a previously defined enumeration tag (E2447)

This declaration is attempting to reference 'ident' as the tag of an enum type, but it has not been so declared.

Correct the name, or rearrange the declarations.

1.1.3.1.536 Undefined label 'identifier' (E2448)

The named label has a `goto` in the function, but no label definition.

1.1.3.1.537 Size of 'identifier' is unknown or zero (E2449)

This identifier was used in a context where its size was needed.

A struct tag might only be declared (the struct not defined yet), or an extern array might be declared without a size.

It's illegal then to have some references to such an item (like sizeof) or to dereference a pointer to this type.

Rearrange your declaration so that the size of 'identifier' is available.

1.1.3.1.538 Undefined structure 'structure' (E2450)

The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file.

This is probably caused by a misspelled structure name or a missing declaration.

1.1.3.1.539 Undefined symbol 'identifier' (E2451)

The named identifier has no declaration.

Possible Causes

- Actual declaration of identifier has been commented out.
- Misspelling, either at this point or at the declaration.
- An error in the declaration of the identifier.
- The header file in which the identifier is declared was not included using `#include`

Tools to help track down the problem:

GREP

1.1.3.1.540 Size of the type 'identifier' is unknown or zero (E2453)

This type was used in a context where its size was needed.

For example, a struct tag might only be declared (the struct not defined yet).

It's illegal then to have some references to such an item (like sizeof) or to dereference a pointer to this type.

Rearrange your declarations so that the size of this type is available.

1.1.3.1.541 Size of the type is unknown or zero (E2452)

This error message indicates that an array of unspecified dimension nested within another structure is initialized and the `-A` (ANSI) switch is on. For example:

```
struct
{
char a[]; //Size of 'a' is unknown or zero
}
b = { "hello" }; //Size of the type is
//unknown or zero
```

1.1.3.1.542 union cannot be a base type (E2454)

A union can't be used as a base type for another class type.

1.1.3.1.543 union cannot have a base type (E2455)

In general, a C++ class can be of union type, but such a class can't be derived from any other class.

1.1.3.1.544 Union member 'member' is of type class with 'constructor' (or destructor, or operator =) (E2456)

A union can't contain members that are of type class with user-defined constructors, destructors, or operator =.

1.1.3.1.545 Delphi style classes must be caught by reference (E2457)

You can only catch a Delphi style object by pointer.

Example

```
void foo(TObject *p)
{
     try
     {
          throw(p);
     }
     catch    (TObject o)    // Error
     {
     }
     catch  (TObject *op)    // OK
     {
     }
}
```

1.1.3.1.546 Delphi classes have to be derived from Delphi classes (E2458)

You cannot derive a Delphi style class from a non-Delphi style class.

Example

```
struct base// base not a Delphi style class
{
intbasemem;
};
struct __declspec(delphiclass) derived : base // or
{
intderivedmem;
};
```

1.1.3.1.547 Delphi style classes must be constructed using operator new (E2459)

Delphi style classes cannot be statically defined. They have to be constructed on the heap.

Example

```
voidfoo(void)
{
     Tobject   o1;     // Error;
     Tobject  *o2 = new TObject();
}
```

1.1.3.1.548 Delphi style classes require exception handling to be enabled (E2460)

If you are using Delphi style classes in your program, you cannot turn off exception handling (compiler option `-x-`) when compiling your source code.

1.1.3.1.549 '%s' requires run-time initialization/finalization (E2461)

This message is issued when a global variable that is declared as __thread (a Win32-only feature) or a static data member of a template class is initialized with a non-constant initial value.

This message is also issued when a global variable that is declared as __thread (a Win32-only feature) or a static data member of a template class has the type class with constructor or destructor.

1.1.3.1.550 'base' is an indirect virtual base class of 'class' (E2463)

You can't create a pointer to a C++ member of a virtual base class.

You have attempted to create such a pointer (either directly, or through a cast) and access an inaccessible member of one of your base classes.

1.1.3.1.551 'virtual' can only be used with member functions (E2464)

A data member has been declared with the `virtual` specifier.

Only member functions can be declared virtual.

For example:

```
class myclass
{
public:
   virtual int a;    //error
};
```

1.1.3.1.552 unions cannot have virtual member functions (E2465)

A union can't have virtual functions as its members.

1.1.3.1.553 void & is not a valid type (E2466)

A reference always refers to an object, but an object cannot have the type void.

Thus, the type void is not allowed.

1.1.3.1.554 'Void function' cannot return a value (E2467)

A function with a return type void contains a return statement that returns a value; for example, an int.

Default = displayed

1.1.3.1.555 Value of type void is not allowed (E2468)

A value of type `void` is really not a value at all, so it can't appear in any context where an actual value is required.

Such contexts include the following:

- the right side of an assignment
- an argument of a function

- the controlling expression of an if, for, or while statement.

1.1.3.1.556 Cannot use tiny or huge memory model with Windows (E2469)

The compiler no longer issues this error.

1.1.3.1.557 CodeGuarded programs must use the large memory model and be targeted for Windows (E2006)

The compiler no longer issues this error.

1.1.3.1.558 Assembler stack overflow (E2375)

The assembler ran out of memory during compilation. Review the portion of code flagged by the error message to ensure that it uses memory correctly.

1.1.3.1.559 The function 'function' is not available (E2269)

You tried to call a function that is known to the evaluator, but which was not present in the program being debugged, for example, an inline function.

1.1.3.1.560 Invalid function call (E2124)

A requested function call failed because the function is not available in the program, a parameter cannot be evaluated, and so on. The evaluator issues this message.

1.1.3.1.561 Invalid 'expression' in scope override (E2213)

The evaluator issues this message when there is an error in a scope override in an expression you are watching or inspecting. You can specify a symbol table, a compilation unit, a source file name, etc. as the scope of the expression, and the message will appear whenever the compiler cannot access the symbol table, compilation unit, or whatever.

1.1.3.1.562 Missing 'identifier' in scope override (E2236)

The syntax of a scope override is somehow incomplete. The evaluator issues this message.

1.1.3.1.563 Pure virtual function called

This is a runtime error. It is generated if the body of a pure virtual function was never generated and somehow the compiler tried to call it.

1.1.3.1.564 String literal not allowed in this context (E2095)

This error message is issued by the evaluator when a string literal appears in a context other than a function call.

1.1.3.1.565 Unexpected termination during compilation [Module Seg#:offset] OR Unexpected termination during linking [Module Seg#:offset]

If either of these errors occur, it indicates a catastrophic failure of the CodeGear tools. You should contact CodeGear to report the problem and to find a potential work around for your specific case. By isolating the test case as well as possible, you will increase the chance for either CodeGear or yourself to find a work around for the problem.

Commonly, compiler failures can be worked around by moving the source code that is currently being compiled. Simple cases might be switching the order of variable declarations, or functions within the source module. Moving the scope and storage of variables also helps in many cases.

For linker failures, you can reduce the amount of debugging information that the linker has to work with. Try compiling only one or two modules with debug information instead of an entire project.

Similarly, switching the order in which object modules are handed to the linker can work around the problem. The IDE hands objects to the linker in the order that they are listed in the project tree. Try moving a source up or down in the list.

1.1.3.1.566 Initializing enumeration with type

You're trying to initialize an enum variable to a different type. For example,

```
enum count { zero, one, two } x = 2;
```

will result in this warning, because 2 is of type int, not type enum count. It is better programming practice to use an enum identifier instead of a literal integer when assigning to or initializing enum types.

This is an error, but is reduced to a warning to give existing programs a chance to work.

1.1.3.1.567 <name> is not a valid identifier

The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there can not be any spaces in the identifier.

1.1.3.1.568 Example for "Temporary used ..." error messages

In this example, function f requires a reference to an int, and c is a char:

```
f(int&);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

1.1.3.1.569 Application is running

The application you tried to run is already running.

For Windows, make sure the message loop of the program has properly terminated.

```
PostQuitMessage(0);
```

1.1.3.1.570 Printf/Scanf floating-point formats not linked

Floating-point formats contain formatting information that is used to manipulate floating-point numbers in certain runtime library

functions, such as scanf() and atof(). Typically, you should avoid linking the floating-point formats (which take up about 1K) unless they are required by your application. However, you must explicitly link the floating-point formats for programs that manipulate fields in a limited and specific way.

Refer to the following list of potential causes (listed from most common to least common) to determine how to resolve this error:

- CAUSE: Floating point set to None. You set the floating-point option to None when it should be set to either Fast or Normal.
- FIX: Set Floating Point to Fast or Normal.
- CAUSE: Either the compiler is over-optimizing or the floating-point formats really do need to be linked. You need the floating-point formats if your program manipulates floats in a limited and specific way. Under certain conditions, the compiler will ignore floating-point usage in scanf(). For example, this may occur when trying to read data into a float variable that is part of an array contained in a structure.
- FIX: Add the following code to one source module:

```
extern _floatconvert;
#pragma extref _floatconvert
```

CAUSE: You forgot to put the address operator & on the scanf variable expression. For example:

```
float foo;
scanf("%f", foo);
```

FIX: Change the code so that the & operator is used where needed. For example, change the above code to the following:

```
float foo;
scanf("%f", &foo);
```

1.1.3.1.571 Ambiguous operators need parentheses (W8000)

(Command-line option to display warning: -wamb)

This warning is displayed whenever two shift, relational, or bitwise-Boolean operators are used together without parentheses.

Also, an addition or subtraction operator that appears without parentheses with a shift operator will produce this warning.

1.1.3.1.572 Possibly incorrect assignment (W8060)

(Command-line option to suppress warning: -w-pia)

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (part of an if, while, or do-while statement).

This is usually a typographical error for the equality operator.

If you want to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly.

For example, this code

```
if (a = b) ...
```

should be rewritten as

```
if ((a = b) != 0) ...
```

1.1.3.1.573 Restarting compile using assembly (W8002)

(Command-line option to suppress warning: -w-asc)

The compiler encountered an asm with no accompanying or #pragma inline statement.

The compile restarts using assembly language capabilities.

Default = On

1.1.3.1.574 Unknown assembler instruction (W8003)

(Command-line option to suppress warning: -w-asm)

The compiler encountered an inline assembly statement with a disallowed opcode or an unknown token. Check the spelling of the opcode or token.

Note: You will get a separate error message from the assembler if you entered illegal assembler source code.

This warning is off by default.

1.1.3.1.575 Throw expression violates exception specification (W8078)

(Command-line option to suppress warning: -w-thr)

This warning happens when you add an exception specification to a function definition and you throw a type in your function body that is not mentioned in your exception specification.

The following program would generate this warning:

```
int foo() throw(char*)  // I promise to only throw char*s
{
    throw 5;  // Oops, I threw an integer
    return 0;
}
```

1.1.3.1.576 Base initialization without a class name is now obsolete (W8052)

(Command-line option to suppress warning: -w-obi)

Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class constructor parameter list.

It is now recommended to include the base class name.

This makes the code much clearer, and is required when you have multiple base classes.

Old way

```
derived::derived(int i) : (i, 10) { ... }
```

New way

```
derived::derived(int i) : base(i, 10) { ... }
```

1.1.3.1.577 Bit fields must be signed or unsigned int (E2117)

(Command-line option to display warning: -wbbf)

In ANSI C, bit fields may not be of type signed char or unsigned char.

When you're not compiling in strict ANSI mode, the compiler allows these constructs, but flags them with this warning.

1.1.3.1.578 Call to function with no prototype (W8064)

(Command-line option to suppress warning: -w-pro)

This message is given if the "Prototypes required" warning is enabled and you call a function without first giving a prototype for

that function.

1.1.3.1.579 Call to function 'function' with no prototype (W8065)

This message is given if the "Prototypes required" warning is enabled and you call function 'function' without first giving a prototype for that function.

1.1.3.1.580 Comparing signed and unsigned values (W8012)

(Command-line option to suppress warning: -w-csu)

Since the ranges of signed and unsigned types are different the result of an ordered comparison of an unsigned and a signed value might have an unexpected result.

Example

```
#pragma warn +csu
boolfoo(unsigned u, int i)
{
return u < i;
}
```

1.1.3.1.581 Constant is long (W8009)

(Command-line option to display warning: -wcln)

The compiler encountered one of the following:

- a decimal constant greater than 32,767 or
- an octal, hexadecimal, or decimal constant greater than 65,535 without a letter l or L following it

The constant is treated as a long.

1.1.3.1.582 Condition is always true OR W8008 Condition is always false (W8008)

(Command-line option to suppress warning: -w-ccc)

Whenever the compiler encounters a constant comparison that (due to the nature of the value being compared) is always true or false, it issues this warning and evaluates the condition at compile time.

For example:

```
  void proc(unsigned x){
    if (x >= 0)        /* always 'true' */
    {
       ...
    }
  }
```

1.1.3.1.583 Continuation character \ found in // comment (W8010)

(Command-line option to suppress warning: -w-com)

This warning message is issued when a C++ // comment is continued onto the next line with backslash line continuation.

The intention is to warn about cases where lines containing source code unintentionally become part of a comment because that comment happened to end in a backslash.

If you get this warning, check carefully whether you intend the line after the // comment to be part of the comment. If you don't, either remove the backslash or put some other character after it. If you do, it's probably better coding style to start the next comment line with // also.

The warning can be disabled altogether with #pragma warn -com.

1.1.3.1.584 'identifier' is declared but never used (W8080)

(Command-line option to display warning: -wuse)

The specified identifier was never used. This message can occur in the case of either local or static variables. It occurs when the source file declares the named local or static variable as part of the block just ending, but the variable was never used.

In the case of local variables, this warning occurs when the compiler encounters the closing brace of the compound statement or function. In the case of static variables, this warning occurs when the compiler encounters the end of the source file.

For example:

```
// Need to compile with -w to make this warning show up!
#pragma option -w
int foo()
{
        int x;
        return 0;
}
```

1.1.3.1.585 Declaration ignored (W8014)

(Command-line option to suppress warning: -w-dig)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

Default = On

1.1.3.1.586 Constant out of range in comparison (W8068)

(Command-line option to suppress warning: -w-rng)

Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type.

For example, comparing an unsigned quantity to -1 makes no sense.

To get an unsigned constant greater than 32,767 (in decimal), you should either

- cast the constant to unsigned--for example, (unsigned) 65535, or
- append a letter u or U to the constant--for example, 65535u.

Whenever this message is issued, the compiler still generates code to do the comparison.

If this code ends up always giving the same result (such as comparing a char expression to 4000), the code will still perform the test.

1.1.3.1.587 Array size for 'delete' ignored (W8016)

(Command-line option to suppress warning: -w-dsz)

The C++ IDE issues this warning when you've specified the array size when deleting an array.

With the new C++ specification, you don't need to make this specification. The compiler ignores this construct.

This warning lets older code compile.

1.1.3.1.588 Division by zero (W8082)

(Command-line option to suppress warning: -w-zdi)

A divide or remainder expression had a literal zero as a divisor.

1.1.3.1.589 Assigning 'type' to 'enumeration' (W8018)

(Command-line option to suppress warning: -w-eas)

Assigning an integer value to an enum type.

This is an error in C++, but is reduced to a warning to give existing programs a chance to work.

1.1.3.1.590 Initializing 'identifier' with 'identifier' (W8006)

(Command-line option to suppress warning: `-w-bei`)

You're trying to initialize an `enum` variable to a different type.

For example, the following initialization will result in this warning, because 2 is of type int, not type enum count:

```
enum count zero, one, two x = 2;
```

It is better programming practice to use an enum identifier instead of a literal integer when assigning to or initializing enum types.

This is an error, but is reduced to a warning to give existing programs a chance to work.

1.1.3.1.591 Superfluous & with function (W8001)

(Command-line option to display warning: -wamp)

An address-of operator (&) is not needed with function name; any such operators are discarded.

1.1.3.1.592 'identifier' is declared as both external and static (W8020)

(Command-line option to suppress warning: -w-ext)

This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration.

The identifier is taken as static.

You should review all declarations for this identifier.

1.1.3.1.593 Hexadecimal value contains more than three digits (W8007)

(Command-line option to suppress warning = -w-big)

Under older versions of C, a hexadecimal escape sequence could contain no more than three digits.

The ANSI standard allows any number of digits to appear as long as the value fits in a byte.

This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as \x00045).

Older versions of C would interpret such a string differently.

1.1.3.1.594 Handler for 'type1' hidden by previous handler for 'type2' (W8021)

(Command-line option to suppress warning: -w-hch)

This warning is issued when a handler for a type 'D' that is derived from type 'B' is specified after a handler for B', since the handler for 'D' will never be invoked.

1.1.3.1.595 Base class 'class1' is also a base class of 'class2' (W8024)

(Command-line option to suppress warning: -w-ibc)

A class inherits from the same base class both directly and indirectly. It is best to avoid this non-portable construct in your program code.

1.1.3.1.596 'function1' hides virtual function 'function2' (W8022)

(Command-line option to suppress warning: -w-hid)

A virtual function in a base class is usually overridden by a declaration in a derived class.

In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.

1.1.3.1.597 Array variable 'identifier' is near (W8023)

(Command-line option to suppress warning: -w-ias)

When you use set the Far Data Threshold option, the compiler automatically makes any global variables that are larger than the threshold size be far.

When the variable is an initialized array with an unspecified size, its total size is not known when the compiler must decide whether to make it near or far, so the compiler makes it near.

The compiler issues this warning if the number of initializers given for the array causes the total variable size to exceed the data size threshold.

If the fact that the compiler made the variable be near causes problems, make the offending variable explicitly far.

To do this, insert the keyword "far" immediately to the left of the variable name in its definition.

1.1.3.1.598 Initialization is only partially bracketed (W8061)

(Command-line option to display warning: `-wpin`)

When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces can be used. This ensures that the compiler's idea and your idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.

1.1.3.1.599 constant member 'identifier' is not initialized (W8038)

(Command-line option to suppress warning: -w-nci)

This C++ class contains a constant member 'member' that doesn't have an initialization.

Note that constant members can be initialized only; they can't be assigned to.

1.1.3.1.600 Integer arithmetic overflow (W8056)

The compiler detected an overflow condition in an integer math expression.

For example:

```
int X = 0xFFFF * 0xFFFF;
```

1.1.3.1.601 Conversion may lose significant digits (W8071)

(Command-line option to display warning: -wsig)

For an assignment operator or some other circumstance, your source file requires a conversion from a larger integral data type to a smaller integral data type where the conversion exists.

Because the integral data type variables don't have the same size, this kind of conversion might alter the behavior of a program.

1.1.3.1.602 Macro definition ignored (W8043)

(Command-line option to suppress warning: -w-nma)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

1.1.3.1.603 Redefinition of 'x' is not identical (W8017)

(Command-line option to suppress warning: -w-dup)

Your source file redefined the macro 'ident' using text that was not exactly the same as the first definition of the macro.

The new text replaces the old.

1.1.3.1.604 User-defined message (W8035)

The error message for which you have requested Help is a user-defined warning.

In C++ code, user-defined messages are introduced by using the `#pragma` message compiler syntax.

`Note:` In addition to messages that you introduce with the `#pragma` message compiler syntax, user-defined warnings can be introduced by third party libraries. Should you require Help about a third party warning, please contact the vendor of the header file that issued the warning.

1.1.3.1.605 Mixing pointers to different 'char' types (W8079)

(Command-line option to display warning: -wucp)

You converted a signed char pointer to an unsigned char pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but it is often harmless.)

1.1.3.1.606 Both return and return with a value used (W8067)

(Command-line option to suppress warning: -w-ret)

The current function has return statements with and without values.

This is legal C, but almost always generates an error.

Possibly a return statement was omitted from the end of the function.

1.1.3.1.607 Negating unsigned value (W8041)

(Command-line option to suppress warning: -w-ngu)

Basically, it makes no sense to negate an unsigned value because the result will still be unsigned.

Example

```
#pragma warn +ngu
unsignedfoo(unsigned u)
{
return-u;
}
```

1.1.3.1.608 Use qualified name to access member type 'identifier' (W8048)

(Command-line option to suppress warning: -w-nst)

In previous versions of the C++ specification, `typedef` and tag names declared inside classes were directly visible in the global scope.

In the latest specification of C++, these names must be prefixed with class`::`qualifier if they are to be used outside of their class scope.

The compiler issues this warning whenever a name is uniquely defined in a single class. The compiler permits this usage without class`::`. This allows older versions of code to compile.

1.1.3.1.609 Use '> >' for nested templates Instead of '>>' (W8049)

(Command-line option to suppress warning: -w-ntd)

Whitespace is required to separate the closing ">" in a nested template name, but since it is an extremely common mistake to leave out the space, the compiler accepts a ">>" with this warning.

1.1.3.1.610 Constructor initializer list ignored (W8039)

(Command-line option to suppress warning: -w-ncl)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

1.1.3.1.611 Function body ignored (W8040)

(Command-line option to suppress warning: -w-nfd)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about

this utility.

1.1.3.1.612 Initializer for object 'x' ignored (W8042)

(Command-line option to suppress warning: -w-nin)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

1.1.3.1.613 Functions containing 'statement' are not expanded inline (W8027)

(Command-line option to suppress warning: -w-inl)

Where:

'statement' can be any of the following:

- Static variables
- Aggregate initializers
- Some return statements
- Local destructors
- Some if statements
- Local classes
- Missing return statements
- Disallowed reserved words listed under "Reserved words" below.

Reserved words

Functions containing any of these reserved words can't be expanded inline, even when specified as inline:

asmexcept

breakfinally

casefor

continuegoto

defaultswitch

dowhile

The function is still perfectly legal, but will be treated as an ordinary static (not global) function.

A copy of the function will appear in each compilation unit where it is called.

Description

If an inline function becomes too complex, the compiler is unable to expand it inline. However, because the function is so complex, expanding it inline is unlikely to provide significant performance enhancements.

For local destructors

You've created an inline function for which the compiler turns off inlining. You can ignore this warning; the function will be generated out of line.

1.1.3.1.614 Functions with exception specifications are not expanded inline (W8026)

Also:Functions taking class by value arguments are not expanded inline

(Command-line option to suppress warning: -w-inl)

Exception specifications are not expanded inline: Check your inline code for lines containing exception specification.

Functions taking class-by-value argument(s) are not expanded inline: When exception handling is enabled, functions that take class arguments by value cannot be expanded inline.

`Note:` Functions taking class parameters by reference are not subject to this restriction.

1.1.3.1.615 #undef directive ignored (W8044)

(Command-line option to suppress warning: -w-nmu)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

1.1.3.1.616 Non-ANSI keyword used: 'keyword' (W8036)

(Command-line option to display warning: -wnak)

A non-ANSI keyword (such as '__fastcall') was used when strict ANSI conformance was requested via the -A option.

1.1.3.1.617 Non-const function 'function' called for const object (W8037)

(Command-line option to suppress warning = -w-ncf)

A non-const member function was called for a const object.

(This is an error, but was reduced to a warning to give existing programs a chance to work.)

1.1.3.1.618 Non-volatile function 'function' called for volatile object (W8051)

(Command-line option to suppress warning: -w-nvf)

In C++, a class member function was called for a volatile object of the class type, but the function was not declared with volatile following the function header. Only a volatile member function can be called for a volatile object.

For example, if you have

```
class c
{
public:
  f() volatile;
  g();
};
volatile c vcvar;
```

it is legal to call vcvar.f(), but not to call vcvar.g().

1.1.3.1.619 Code has no effect (W8019)

(Command-line option to suppress warning: -w-eff)

This warning is issued when the compiler encounters a statement with some operators that have no effect.

For example, the statement

```
a + b;
```

has no effect on either variable.

The operation is unnecessary and probably indicates a bug.

1.1.3.1.620 Parameter 'parameter' is never used (W8057)

(Command-line option to suppress warning: -w-par)

The named parameter, declared in the function, was never used in the body of the function.

This might or might not be an error and is often caused by misspelling the parameter.

This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function.

The parameter is masked by the automatic variable and remains unused.

1.1.3.1.621 Cannot create pre-compiled header: 'reason' (W8058)

(Command-line option to suppress warning: -w-pch)

This warning is issued when pre-compiled headers are enabled but the compiler could not generate one, for one of the following reasons:

ReasonExplanation

write failedThe compiler could not write to the pre-compiled header file. This occurs if you specified an invalid location to cache precompiled headers or if the disk is full.

code in headerOne of the headers contained a non-inline function body.

initialized data in headerOne of the headers contained a global variable definition (in C, a global variable with an initializer; in C++ any variable not declared as 'extern').

header incompleteThe pre-compiled header ended in the middle of a declaration, for example, inside a class definition (this often happens when there is a missing closing brace in a header file).

1.1.3.1.622 Pragma option pop with no matching option push (W8046)

The compiler encountered a #pragma option pop before a previous #pragma option push, or in the case of nesting, there are more occurrences of #pragma option pop than of #pragma option push.

1.1.3.1.623 Function should return a value (W8070)

(Command-line option to suppress warning: `-w-rvl`)

This function was declared (maybe implicitly) to return a value.

The compiler found a return statement without a return value, or it reached the end of the function without finding a return

statement.

Either return a value or change the function declaration to return void.

1.1.3.1.624 Declaration of static function function ignored (W8047)

(Command-line option to suppress warning: -w-nsf)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

1.1.3.1.625 No type OBJ file present. Disabling external types option. (W8050)

(Command-line option to suppress warning: -w-nto)

A precompiled header file references a type object file, but the type object file cannot be found. This is not a fatal problem but will make your object files larger than necessary.

1.1.3.1.626 'ident' is obsolete (W8053)

(Command-line option to suppress warning: -w-obs)

Issues a warning upon usage for any "C" linkage function that has been specified. This will warn about functions that are "obsolete".

Here's an example of it's usage:

```
#ifdef __cplusplus
extern "C" {
#endif
void my_func(void);
#ifdef __cplusplus
}
#endif
#pragma obsolete my_func
main()
{
my_func();   // Generates warning about obsolete function
}
```

1.1.3.1.627 Style of function definition is now obsolete (W8054)

(Command-line option to suppress warning = -w-ofp)

In C++, this old C style of function definition is illegal:

```
int func(p1, p2) int p1, p2; { /* ... */ }
```

This practice might not be allowed by other C++ compilers.

1.1.3.1.628 Path 'path' and filename 'filename' exceed maximum size of 'n' (W8103)

(Command-line option to display warning: -wstv)

In looking up include files, the C++ compiler has encountered a file whose path and filename contain more characters than are allowed in the Windows maximum. Rename the paths and filenames that you can, and shorten their names wherever possible.

1.1.3.1.629 Ill-formed pragma (W8025)

(Command-line option to suppress warning: -w-ill)

A pragma does not match one of the pragmas expected by the compiler.

1.1.3.1.630 Overloaded prefix operator 'operator' used as a postfix operator (W8063)

(Command-line option to suppress warning: -w-pre)

The C++ specification allows you to overload both the prefix and postfix versions of the ++ and -- operators.

Whenever the prefix operator is overloaded, but is used in a postfix context, the compiler uses the prefix operator and issues this warning.

This allows older code to compile.

1.1.3.1.631 Declare 'type' prior to use in prototype (W8015)

(Command-line option to suppress warning: -w-dpu)

When a function prototype refers to a structure type that has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype.

For example,

```
int func(struct s *ps);  struct s  /* ... */ ;
```

Because there is no "struct s" in scope at the prototype for func, the type of parameter ps is pointer to undefined struct s, and is not the same as the "struct s" that is later declared.

This will result in later warning and error messages about incompatible types, which would be very mysterious without this warning message.

To fix the problem, you can move the declaration for "struct s" ahead of any prototype that references it, or add the incomplete type declaration "struct s;" ahead of any prototype that references "structs".

If the function parameter is a struct, rather than a pointer to struct, the incomplete declaration is not sufficient.

You must then place the struct declaration ahead of the prototype.

1.1.3.1.632 Nonportable pointer conversion (W8069)

(Command-line option to suppress warning: `-w-rpt`)

A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same.

Use an explicit cast if this is what you really meant to do.

1.1.3.1.633 Previous options and warnings not restored (W8062)

The compiler didn't encounter a #pragma option pop after a previous #pragma option push, or in the case of nesting, there are more occurrences of #pragma option push than of #pragma option pop.

1.1.3.1.634 Unreachable code (W8066)

(Command-line option to suppress warning: -w-rch)

A break, continue, goto, or return statement was not followed by a label or the end of a loop or function.

The compiler checks while, do, and for loops with a constant test condition, and attempts to recognize loops that can't fall through.

1.1.3.1.635 Temporary used for parameter '???' (W8029)

(Command-line option to suppress warning: -w-lvc)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

In the following example, function f requires a reference to an int, and c is a char:

```
f(int &amp);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

1.1.3.1.636 Temporary used for parameter 'parameter' OR W8029 Temporary used for parameter 'number' OR W8030 Temporary used for parameter 'parameter' in call to 'function' OR W8032 Temporary used for parameter 'number' in call to 'function' (W8031)

(Command-line option to suppress warning: -w-lvc)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type.

If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

1.1.3.1.637 Temporary used for parameter 2 in call to '???' (W8032)

(Command-line option to suppress warning: -w-lvc)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

In the following example, function f requires a reference to an int, and c is a char:

```
f(int &amp);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

1.1.3.1.638 Temporary used to initialize 'identifier' (W8028)

(Command-line option to suppress warning: -w-lin)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type.

If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

Example

In this example, function f requires a reference to an int, and c is a char:

```
f(int&);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

1.1.3.1.639 Possible overflow in shift operation (W8055)

The compiler detects cases where the number of bits shifted over is larger than the number of bits in the affected variable; for example:

```
char c;  c >> 16;
```

1.1.3.1.640 Structure packing size has changed (W8059)

(Command-line option to suppress warning: -w-pck)

This warning message is issued when the structure alignment is different after including a file than it was before including that file.

The intention is to warn you about cases where an include file changes structure packing, but by mistake doesn't restore the original setting at the end. If this is intentional, you can give a #pragma nopackwarning directive at the end of an include file to disable the warning for this file.

The warning can be disabled altogether by #pragma warn -pck.

1.1.3.1.641 Structure passed by value (W8074)

(Command-line option to display warning: -wstv)

This warning is generated any time a structure is passed by value as an argument.

It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument.

Because structures can be passed by value, this omission is acceptable.

This warning provides a way for the compiler to warn you of this mistake.

1.1.3.1.642 Suspicious pointer arithmetic (W8072)

This message indicates an unintended side effect to the pointer arithmetic (or array indexing) found in an expression.

Example

```
#pragma warn +spa
intarray[10];
intfoo(__int64 index)
{
returnarray[index];
}
```

The value of index is 64 bits wide while the address of array is only 32 bits wide.

1.1.3.1.643 Nonportable pointer comparison (W8011)

(Command-line option to suppress warning: -w-cpt)

Your source file compared a pointer to a non-pointer other than the constant 0.

You should use a cast to suppress this warning if the comparison is proper.

1.1.3.1.644 Suspicious pointer conversion (W8075)

(Command-line option to suppress warning: -w-sus)

The compiler encountered some conversion of a pointer that caused the pointer to point to a different type.

You should use a cast to suppress this warning if the conversion is proper.

A common cause of this warning is when the C compiler converts a function pointer of one type to another (the C++ compiler generates an error when asked to do that). It can be suppressed by doing a typecast. Here is a common occurrence of it for Windows programmers:

```
#define STRICT
#include <windows.h>
LPARAM _export WndProc( HWND , UINT , WPARAM , LPARAM );
test() {
   WNDCLASS wc;
   wc.lpfnWndProc = WndProc;  //warning
}
```

It is suppressed by making the assignment to lpfnWndProc as follows:

```
wc.lpfnWndProc = ( WNDPROC ) WndProc;
```

1.1.3.1.645 No declaration for function 'function' (W8045)

(Command-line option to display warning: -wnod)

This message is given if you call a function without first declaring that function.

In C, you can declare a function without presenting a prototype, as in

```
int func();
```

In C++, every function declaration is also a prototype; this example is equivalent to

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

The declaration can be either classic or modern (prototype) style.

1.1.3.1.646 Undefined structure 'structure' (W8073)

(Command-line option to display warning = `-wstu`)

Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure.

This is probably caused by a misspelled structure name or a missing declaration.

1.1.3.1.647 Possible use of 'identifier' before definition (W8013)

(Command-line option to display warning: -wdef)

Your source file used the variable 'identifier' in an expression before it was assigned a value.

The compiler uses a simple scan of the program to determine this condition.

If the use of a variable occurs physically before any assignment, this warning will be generated.

Of course, the actual flow of the program can assign the value before the program uses it.

1.1.3.1.648 'identifier' is assigned a value that is never used (W8004)

(Command-line option to suppress warning: -w-aus)

The variable appears in an assignment, but is never used anywhere else in the function just ending.

The warning is indicated only when the compiler encounters the closing brace.

The `#pragma warn -aus` switch has function-level granularity. You cannot turn off this warning for individual variables within a function; it is either off or on for the whole function.

1.1.3.1.649 Conversion to 'type' will fail for members of virtual base 'class' (W8033)

(Command-line option to suppress warning: -w-mpc)

This warning is issued only if the `-Vv` option is in use.

The warning may be issued when a member pointer to one type is cast to a member pointer of another type and the class of the converted member pointer has virtual bases.

Encountering this warning means that at runtime, if the member pointer conversion cannot be completed, the result of the cast will be a NULL member pointer.

1.1.3.1.650 Maximum precision used for member pointer type 'type' (W8034)

(Command-line option to suppress warning: -w-mpd)

When a member pointer type is declared, its class has not been fully defined, and the `-Vmd` option has been used, the compiler has to use the most general (and the least efficient) representation for that member pointer type. This can cause less efficient code to be generated (and make the member pointer type unnecessarily large), and can also cause problems with separate

compilation; see the `-Vm` compiler switch for details.

1.1.3.1.651 Void functions may not return a value (W8081)

(Command-line option to suppress warning: -w-voi)

Your source file declared the current function as returning void, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.

Example

```
// This HAS to be in a "C" file.  In a "C++" file this would be an error
void foo()
{
        return 0;       // Can't return a value from a void function
}
```

Index

+

<

=

B

'

B

C

'

C

'

C

'

D

E

'

E

F

'

F

'

F

'

F

G

H

I

I

I

J

K

L

M

'

M

'

M

N

'

N

Q

R

S

'

S

'

U

V