Contents

C Language Reference

12/22/2021 • 2 minutes to read • Edit Online

The *C Language Reference* describes the C programming language as implemented in Microsoft C. The book's organization is based on the ANSI C standard (sometimes referred to as C89) with additional material on the Microsoft extensions to the ANSI C standard.

- Organization of the C Language Reference

For additional reference material on C++ and the preprocessor, see:

- C++ Language Reference
- Preprocessor Reference

Compiler and linker options are documented in the C/C++ Building Reference.

See also

C++ Language Reference

Organization of the C Language Reference

12/22/2021 • 2 minutes to read • Edit Online

- Elements of C
- Program Structure
- Declarations and Types
- Expressions and Assignments
- Statements
- Functions
- C Language Syntax Summary
- Implementation-Defined Behavior

See also

C Language Reference

Scope of this Manual

12/22/2021 • 2 minutes to read • Edit Online

C is a flexible language that leaves many programming decisions up to you. In keeping with this philosophy, C imposes few restrictions in matters such as type conversion. Although this characteristic of the language can make your programming job easier, you must know the language well to understand how programs will behave. This book provides information on the C language components and the features of the Microsoft implementation. The syntax for the C language is from ANSI X3.159-1989, *American National Standard for Information Systems - Programming Language - C* (hereinafter called the ANSI C standard), although it is not part of the ANSI C standard. C Language Syntax Summary provides the syntax and a description of how to read and use the syntax definitions.

This book does not discuss programming with C++. See C++ Language Reference for information about the C++ language.

See also

Organization of the C Language Reference

ANSI Conformance

12/22/2021 • 2 minutes to read • Edit Online

Microsoft C conforms to the standard for the C language as set forth in the 9899:1990 edition of the ANSI C standard.

Microsoft extensions to the ANSI C standard are noted in the text and syntax of this book as well as in the online reference. Because the extensions are not a part of the ANSI C standard, their use may restrict portability of programs between systems. By default, the Microsoft extensions are enabled. To disable the extensions, specify the /Za compiler option. With /Za, all non-ANSI code generates errors or warnings.

See also

Organization of the C Language Reference

Elements of C

12/22/2021 • 2 minutes to read • Edit Online

This section describes the elements of the C programming language, including the names, numbers, and characters used to construct a C program. The ANSI C syntax labels these components tokens.

This section explains how to define tokens and how the compiler evaluates them.

The following topics are discussed:

- Tokens
- Comments
- Keywords
- Identifiers
- Constants
- String literals
- Punctuation and special characters

The section also includes reference tables for Trigraphs, Limits on Floating-Point Constants, C and C++ Integer Limits, and Escape Sequences.

Operators are symbols (both single characters and character combinations) that specify how values are to be manipulated. Each symbol is interpreted as a single unit, called a token. For more information, see Operators.

See also

C Language Reference

C Tokens

12/22/2021 • 2 minutes to read • Edit Online

In a C source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements.

Syntax

token: *keyword*

identifier

constant

string-literal

operator

punctuator

> **NOTE**
>
> See the introduction to C Language Syntax Summary for an explanation of the ANSI syntax conventions.

The keywords, identifiers, constants, string literals, and operators described in this section are examples of tokens. Punctuation characters such as brackets (**[ ]**), braces (**{ }**), parentheses (**( )**), and commas (**,**) are also tokens.

See also

Elements of C

White-Space Characters

12/22/2021 • 2 minutes to read • Edit Online

Space, tab, line feed (newline), carriage return, form feed, and vertical tab characters are called "white-space characters" because they serve the same purpose as the spaces between words and lines on a printed page — they make reading easier. Tokens are delimited (bounded) by white-space characters and by other tokens, such as operators and punctuation. When parsing code, the C compiler ignores white-space characters unless you use them as separators or as components of character constants or string literals. Use white-space characters to make a program more readable. Note that the compiler also treats comments as white space.

See also

C Tokens

C Comments

12/22/2021 • 2 minutes to read • Edit Online

A "comment" is a sequence of characters beginning with a forward slash/asterisk combination (**/***) that is treated as a single white-space character by the compiler and is otherwise ignored. A comment can include any combination of characters from the representable character set, including newline characters, but excluding the "end comment" delimiter (***/**). Comments can occupy more than one line but cannot be nested.

Comments can appear anywhere a white-space character is allowed. Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens. The compiler ignores the characters in the comment.

Use comments to document your code. This example is a comment accepted by the compiler:

```
/* Comments can contain keywords such as
   for and while without generating errors. */
```

Comments can appear on the same line as a code statement:

```
printf( "Hello\n" );  /* Comments can go here */
```

You can choose to precede functions or program modules with a descriptive comment block:

```
/* MATHERR.C illustrates writing an error routine
* for math functions.
*/
```

Since comments cannot contain nested comments, this example causes an error:

```
/* Comment out this routine for testing

   /* Open file */
    fh = _open( "myfile.c", _O_RDONLY );
    .
    .
    .
*/
```

The error occurs because the compiler recognizes the first `*/`, after the words `Open file`, as the end of the comment. It tries to process the remaining text and produces an error when it finds the `*/` outside a comment.

While you can use comments to render certain lines of code inactive for test purposes, the preprocessor directives `#if` and `#endif` and conditional compilation are a useful alternative for this task. For more information, see Preprocessor Directives in the *Preprocessor Reference*.

Microsoft Specific

The Microsoft compiler also supports single-line comments preceded by two forward slashes (**//**). If you compile with /Za (ANSI standard), these comments generate errors. These comments cannot extend to a second line.

```
// This is a valid comment
```

Comments beginning with two forward slashes (**//**) are terminated by the next newline character that is not preceded by an escape character. In the next example, the newline character is preceded by a backslash (****), creating an "escape sequence." This escape sequence causes the compiler to treat the next line as part of the previous line. (For more information, see Escape Sequences.)

```
// my comment \
    i++;
```

Therefore, the `i++;` statement is commented out.

The default for Microsoft C is that the Microsoft extensions are enabled. Use /Za to disable these extensions.

END Microsoft Specific

See also

C Tokens

Evaluation of Tokens

12/22/2021 • 2 minutes to read • Edit Online

When the compiler interprets tokens, it includes as many characters as possible in a single token before moving on to the next token. Because of this behavior, the compiler may not interpret tokens as you intended if they are not properly separated by white space. Consider the following expression:

```
i+++j
```

In this example, the compiler first makes the longest possible operator (`++`) from the three plus signs, then processes the remaining plus sign as an addition operator (`+`). Thus, the expression is interpreted as `(i++) + (j)`, not `(i) + (++j)`. In this and similar cases, use white space and parentheses to avoid ambiguity and ensure proper expression evaluation.

Microsoft Specific

The C compiler treats a CTRL+Z character as an end-of-file indicator. It ignores any text after CTRL+Z.

END Microsoft Specific

See also

C Tokens

C Keywords

12/22/2021 • 2 minutes to read • Edit Online

Keywords are words that have special meaning to the C compiler. In translation phases 7 and 8, an identifier can't have the same spelling and case as a C keyword. For more information, see translation phases in the *Preprocessor Reference*. For more information on identifiers, see Identifiers.

Standard C keywords

The C language uses the following keywords:

`auto`
`break`
`case`
`char`
`const`
`continue`
`default`
`do`
`double`
`else`
`enum`

`extern`
`float`
`for`
`goto`
`if`
`inline` [1, a]
`int`
`long`
`register`
`restrict` [1, a]
`return`

`short`
`signed`
`sizeof`
`static`
`struct`
`switch`
`typedef`
`union`
`unsigned`
`void`
`volatile`

`while`
`_Alignas` [2, a]

`_Alignof` [2, a]
`_Atomic` [2, b]
`_Bool` [1, a]
`_Complex` [1, b]
`_Generic` [2, a]
`_Imaginary` [1, b]
`_Noreturn` [2, a]
`_Static_assert` [2, a]
`_Thread_local` [2, b]

[1] Keywords introduced in ISO C99.

[2] Keywords introduced in ISO C11.

[a] Starting in Visual Studio 2019 version 16.8, these keywords are supported in code compiled as C when the `/std:c11` or `/std:c17` compiler options are specified.

[b] Starting in Visual Studio 2019 version 16.8, these keywords are recognized but not supported by the compiler in code compiled as C when the `/std:c11` or `/std:c17` compiler options are specified.

You can't redefine keywords. However, you can specify text to replace keywords before compilation by using C preprocessor directives.

Microsoft-specific C keywords

The ANSI and ISO C standards allow identifiers with two leading underscores to be reserved for compiler implementations. The Microsoft convention is to precede Microsoft-specific keyword names with double underscores. These words can't be used as identifier names. For a description of the rules for naming identifiers, including the use of double underscores, see Identifiers.

The following keywords and special identifiers are recognized by the Microsoft C compiler:

`__asm` [5]
`__based` [3, 5]
`__cdecl` [5]
`__declspec` [5]
`__except` [5]
`__fastcall`
`__finally` [5]

`__inline` [5]
`__int16` [5]
`__int32` [5]
`__int64` [5]
`__int8` [5]
`__leave` [5]
`__restrict`

`__stdcall` [5]
`__try` [5]
`dllexport` [4]
`dllimport` [4]
`naked` [4]
`static_assert` [6]
`thread` [4]

[3] The `__based` keyword has limited uses for 32-bit and 64-bit target compilations.

[4] These are special identifiers when used with `__declspec`; their use in other contexts is unrestricted.

[5] For compatibility with previous versions, these keywords are available both with two leading underscores and a single leading underscore when Microsoft extensions are enabled.

[6] If you don't include <assert.h>, the Microsoft Visual C compiler maps `static_assert` to the C11 `_Static_assert` keyword.

Microsoft extensions are enabled by default. To assist in creating portable code, you can disable Microsoft extensions by specifying the /Za (Disable language extensions) option during compilation. When you use this option, some Microsoft-specific keywords are disabled.

When Microsoft extensions are enabled, you can use the keywords listed above in your programs. To conform to the language standard, most of these keywords are prefaced by a double underscore. The four exceptions, `dllexport`, `dllimport`, `naked`, and `thread`, are used only with `__declspec` and don't require a leading double underscore. For backward compatibility, single-underscore versions of the rest of the keywords are supported.

See also

Elements of C

C Identifiers

12/22/2021 • 3 minutes to read • Edit Online

"Identifiers" or "symbols" are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You cannot use keywords (either C or Microsoft) as identifiers; they are reserved for special use. You create an identifier by specifying it in the declaration of a variable, type, or function. In this example, `result` is an identifier for an integer variable, and `main` and `printf` are identifier names for functions.

```
#include <stdio.h>

int main()
{
    int result;

    if ( result != 0 )
        printf_s( "Bad file handle\n" );
}
```

Once declared, you can use the identifier in later program statements to refer to the associated value.

A special kind of identifier, called a statement label, can be used in `goto` statements. (Declarations are described in Declarations and Types Statement labels are described in The goto and Labeled Statements.)

Syntax

identifier:
  nondigit
  identifier nondigit
  identifier digit

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

digit: one of
  0 1 2 3 4 5 6 7 8 9

The first character of an identifier name must be a `nondigit` (that is, the first character must be an underscore or an uppercase or lowercase letter). ANSI allows six significant characters in an external identifier's name and 31 for names of internal (within a function) identifiers. External identifiers (ones declared at global scope or declared with storage class `extern`) may be subject to additional naming restrictions because these identifiers have to be processed by other software such as linkers.

Microsoft Specific

Although ANSI allows 6 significant characters in external identifier names and 31 for names of internal (within a function) identifiers, the Microsoft C compiler allows 247 characters in an internal or external identifier name. If you aren't concerned with ANSI compatibility, you can modify this default to a smaller or larger number using the /H (restrict length of external names) option.

END Microsoft Specific

The C compiler considers uppercase and lowercase letters to be distinct characters. This feature, called "case sensitivity," enables you to create distinct identifiers that have the same spelling but different cases for one or more of the letters. For example, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

Microsoft Specific

Do not select names for identifiers that begin with two underscores or with an underscore followed by an uppercase letter. The ANSI C standard allows identifier names that begin with these character combinations to be reserved for compiler use. Identifiers with file-level scope should also not be named with an underscore and a lowercase letter as the first two letters. Identifier names that begin with these characters are also reserved. By convention, Microsoft uses an underscore and an uppercase letter to begin macro names and double underscores for Microsoft-specific keyword names. To avoid any naming conflicts, always select identifier names that do not begin with one or two underscores, or names that begin with an underscore followed by an uppercase letter.

END Microsoft Specific

The following are examples of valid identifiers that conform to either ANSI or Microsoft naming restrictions:

```
j
count
temp1
top_of_page
skip12
LastNum
```

Microsoft Specific

Although identifiers in source files are case sensitive by default, symbols in object files are not. Microsoft C treats identifiers within a compilation unit as case sensitive.

The Microsoft linker is case sensitive. You must specify all identifiers consistently according to case.

The "source character set" is the set of legal characters that can appear in source files. For Microsoft C, the source set is the standard ASCII character set. The source character set and execution character set include the ASCII characters used as escape sequences. See Character Constants for information about the execution character set.

END Microsoft Specific

An identifier has "scope," which is the region of the program in which it is known, and "linkage," which determines whether the same name in another scope refers to the same identifier. These topics are explained in Lifetime, Scope, Visibility, and Linkage.

See also

Elements of C

Multibyte and Wide Characters

12/22/2021 • 2 minutes to read • Edit Online

A multibyte character is a character composed of sequences of one or more bytes. Each byte sequence represents a single character in the extended character set. Multibyte characters are used in character sets such as Kanji.

Wide characters are multilingual character codes that are always 16 bits wide. The type for character constants is `char`; for wide characters, the type is `wchar_t`. Since wide characters are always a fixed size, using wide characters simplifies programming with international character sets.

The wide-character-string literal `L"hello"` becomes an array of six integers of type `wchar_t`.

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

The Unicode specification is the specification for wide characters. The run-time library routines for translating between multibyte and wide characters include `mbstowcs`, `mbtowc`, `wcstombs`, and `wctomb`.

See also

C Identifiers

Trigraphs

12/22/2021 • 2 minutes to read • Edit Online

The source character set of C source programs is contained within the 7-bit ASCII character set but is a superset of the ISO 646-1983 Invariant Code Set. Trigraph sequences allow C programs to be written using only the ISO (International Standards Organization) Invariant Code Set. Trigraphs are sequences of three characters (introduced by two consecutive question marks) that the compiler replaces with their corresponding punctuation characters. You can use trigraphs in C source files with a character set that does not contain convenient graphic representations for some punctuation characters.

C++17 removes trigraphs from the language. Implementations may continue to support trigraphs as part of the implementation-defined mapping from the physical source file to the *basic source character set*, though the standard encourages implementations not to do so. Through C++14, trigraphs are supported as in C.

Visual C++ continues to support trigraph substitution, but it's disabled by default. For information on how to enable trigraph substitution, see `/Zc:trigraphs` (Trigraphs Substitution).

The following table shows the nine trigraph sequences. All occurrences in a source file of the punctuation characters in the first column are replaced with the corresponding character in the second column.

Trigraph Sequences

TRIGRAPH	PUNCTUATION CHARACTER
`??=`	`#`
`??(`	`[`
`??/`	`\`
`??)`	`]`
`??'`	`^`
`??<`	`{`
`??!`	`|`
`??>`	`}`
`??-`	`~`

A trigraph is always treated as a single source character. The translation of trigraphs takes place in the first translation phase, before the recognition of escape characters in string literals and character constants. Only the nine trigraphs shown in the above table are recognized. All other character sequences are left untranslated.

The character escape sequence, `\?`, prevents the misinterpretation of trigraph-like character sequences. (For information about escape sequences, see Escape Sequences.) For example, if you attempt to print the string `What??!` with this `printf` statement

```
printf( "What??!\n" );
```

the string printed is `What|` because `??!` is a trigraph sequence that is replaced with the `|` character. Write the statement as follows to correctly print the string:

```
printf( "What?\?!\n" );
```

In this `printf` statement, a backslash escape character in front of the second question mark prevents the misinterpretation of `??!` as a trigraph.

See also

`/Zc:trigraphs` (Trigraphs Substitution)
C Identifiers

C Constants

12/22/2021 • 2 minutes to read • Edit Online

A *constant* is a number, character, or character string that can be used as a value in a program. Use constants to represent floating-point, integer, enumeration, or character values that cannot be modified.

Syntax

`constant` :
  `floating-point-constant`
  `integer-constant`
  `enumeration-constant`
  `character-constant`

Constants are characterized by having a value and a type. Floating-point, integer, and character constants are discussed in the next three sections. Enumeration constants are described in Enumeration Declarations.

See also

Elements of C

C Floating-Point Constants

12/22/2021 • 2 minutes to read • Edit Online

A "floating-point constant" is a decimal number that represents a signed real number. The representation of a signed real number includes an integer portion, a fractional portion, and an exponent. Use floating-point constants to represent floating-point values that can't be changed.

Syntax

`floating-point-constant` :
  `fractional-constant` `exponent-part` opt `floating-suffix` opt
  `digit-sequence` `exponent-part` `floating-suffix` opt

`fractional-constant` :
  `digit-sequence` opt . `digit-sequence`
  `digit-sequence` .

`exponent-part` :
  e `sign` opt `digit-sequence`
  E `sign` opt `digit-sequence`

`sign` : one of
  `+` `-`

`digit-sequence` :
  `digit`
  `digit-sequence` `digit`

`floating-suffix` : one of
  `f` `l` `F` `L`

You can omit either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion), but not both. You may leave out the decimal point only if you include an exponent. No white-space characters can separate the digits or characters of the constant.

The following examples illustrate some forms of floating-point constants and expressions:

```
15.75
1.575E1   /* = 15.75   */
1575e-2   /* = 15.75   */
-2.5e-3   /* = -0.0025 */
25E-4     /* =  0.0025 */
```

Floating-point constants are positive unless they're preceded by a minus sign (`-`). In this case, the minus sign is treated as a unary arithmetic negation operator. Floating-point constants have type `float`, `double`, or `long double`.

A floating-point constant without an `f`, `F`, `l`, or `L` suffix has type `double`. If the letter `f` or `F` is the suffix, the constant has type `float`. If suffixed by the letter `l` or `L`, it has type `long double`. For example:

```
10.0L  /* Has type long double  */
10.0   /* Has type double       */
10.0F  /* Has type float        */
```

The Microsoft C compiler internally represents `long double` the same as type `double`. However, the types are distinct. See Storage of basic types for information about type `double`, `float`, and `long double`.

You can omit the integer portion of the floating-point constant, as shown in the following examples. The number 0.75 can be expressed in many ways, including the following examples:

```
.0075e2
0.075e1
.075e1
75e-2
```

See also

C constants

Limits on Floating-Point Constants

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

Limits on the values of floating-point constants are given in the following table. The header file FLOAT.H contains this information.

Limits on Floating-Point Constants

CONSTANT	MEANING	VALUE
FLT_DIG DBL_DIG LDBL_DIG	Number of digits, *q*, such that a floating-point number with *q* decimal digits can be rounded into a floating-point representation and back without loss of precision.	6 15 15
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	Smallest positive number *x*, such that *x* + 1.0 is not equal to 1.0	1.192092896e-07F 2.2204460492503131e-016 2.2204460492503131e-016
FLT_GUARD		0
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	Number of digits in the radix specified by **FLT_RADIX** in the floating-point significand. The radix is 2; hence these values specify bits.	24 53 53
FLT_MAX DBL_MAX LDBL_MAX	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 308
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	Maximum integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	128 1024 1024
FLT_MIN DBL_MIN LDBL_MIN	Minimum positive value.	1.175494351e-38F 2.2250738585072014e-308 2.2250738585072014e-308
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that number is a representable floating-point number.	-37 -307 -307
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	Minimum negative integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	-125 -1021 -1021
FLT_NORMALIZE		0

CONSTANT	MEANING	VALUE
FLT_RADIX _DBL_RADIX _LDBL_RADIX	Radix of exponent representation.	2 2 2
FLT_ROUNDS _DBL_ROUNDS _LDBL_ROUNDS	Rounding mode for floating-point addition.	1 (near) 1 (near) 1 (near)

Note that the information in the above table may differ in future implementations.

END Microsoft Specific

See also

C Floating-Point Constants

C Integer Constants

12/22/2021 • 2 minutes to read • Edit Online

An *integer constant* is a decimal (base 10), octal (base 8), or hexadecimal (base 16) number that represents an integral value. Use integer constants to represent integer values that cannot be changed.

Syntax

integer-constant:
  decimal-constant integer-suffix$_{opt}$
  octal-constant integer-suffix$_{opt}$
  hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
  nonzero-digit
  decimal-constant digit

octal-constant:
  0
  octal-constant octal-digit

hexadecimal-constant:
  hexadecimal-prefix hexadecimal-digit
  hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of
  0x 0X

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

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

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

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  unsigned-suffix long-long-suffix
  unsigned-suffix 64-bit-integer-suffix
  long-suffix unsigned-suffix$_{opt}$
  long-long-suffix unsigned-suffix$_{opt}$
  64-bit-integer-suffix

unsigned-suffix: one of
  u U

long-suffix: one of
  l L

long-long-suffix: one of
  ll LL

64-bit-integer-suffix: one of
  i64 I64

The **i64** and **I64** suffixes are Microsoft-specific.

Integer constants are positive unless they are preceded by a minus sign (-). The minus sign is interpreted as the unary arithmetic negation operator. (See Unary Arithmetic Operators for information about this operator.)

If an integer constant begins with **0x** or **0X**, it is hexadecimal. If it begins with the digit **0**, it is octal. Otherwise, it is assumed to be decimal.

The following integer constants are equivalent:

```
28
0x1C   /* = Hexadecimal representation for decimal 28 */
034    /* = Octal representation for decimal 28 */
```

No white-space characters can separate the digits of an integer constant. These examples show some valid decimal, octal, and hexadecimal constants.

```
    /* Decimal Constants */
    int                dec_int    = 28;
    unsigned           dec_uint   = 4000000024u;
    long               dec_long   = 2000000022l;
    unsigned long      dec_ulong  = 4000000000ul;
    long long          dec_llong  = 9000000000LL;
    unsigned long long dec_ullong = 900000000001ull;
    __int64            dec_i64    = 9000000000002I64;
    unsigned __int64   dec_ui64   = 90000000000004ui64;

    /* Octal Constants */
    int                oct_int    = 024;
    unsigned           oct_uint   = 04000000024u;
    long               oct_long   = 02000000022l;
    unsigned long      oct_ulong  = 04000000000UL;
    long long          oct_llong  = 044000000000000ll;
    unsigned long long oct_ullong = 044400000000000001Ull;
    __int64            oct_i64    = 04444000000000000002i64;
    unsigned __int64   oct_ui64   = 04444000000000000004uI64;

    /* Hexadecimal Constants */
    int                hex_int    = 0x2a;
    unsigned           hex_uint   = 0XA0000024u;
    long               hex_long   = 0x20000022l;
    unsigned long      hex_ulong  = 0XA0000021uL;
    long long          hex_llong  = 0x8a000000000000ll;
    unsigned long long hex_ullong = 0x8A40000000000010uLL;
    __int64            hex_i64    = 0x4a44000000000020I64;
    unsigned __int64   hex_ui64   = 0x8a44000000000040Ui64;
```

See also

C Constants

Integer Types

12/22/2021 • 2 minutes to read • Edit Online

Every integer constant is given a type based on its value and the way it's expressed. You can force any integer constant to type `long` by appending the letter `l` or `L` to the end of the constant; you can force it to be type `unsigned` by appending `u` or `U` to the value. The lowercase letter `l` can be confused with the digit 1 and should be avoided. Some forms of `long` integer constants follow:

```
/* Long decimal constants */
10L
79L

/* Long octal constants */
012L
0115L

/* Long hexadecimal constants */
0xaL or 0xAL
0X4fL or 0x4FL

/* Unsigned long decimal constant */
776745UL
778866LU
```

The type you assign to a constant depends on the value the constant represents. A constant's value must be in the range of representable values for its type. A constant's type determines which conversions are performed when the constant is used in an expression or when the minus sign (`-`) is applied. This list summarizes the conversion rules for integer constants.

- The type for a decimal constant without a suffix is either `int`, `long int`, or `unsigned long int`. The first of these three types in which the constant's value can be represented is the type assigned to the constant.
- The type assigned to octal and hexadecimal constants without suffixes is `int`, `unsigned int`, `long int`, or `unsigned long int` depending on the size of the constant.
- The type assigned to constants with a `u` or `U` suffix is `unsigned int` or `unsigned long int` depending on their size.
- The type assigned to constants with an `l` or `L` suffix is `long int` or `unsigned long int` depending on their size.
- The type assigned to constants with a `u` or `U` and an `l` or `L` suffix is `unsigned long int`.

See also

C Integer Constants

C and C++ Integer Limits

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The limits for integer types in C and C++ are listed in the following table. These limits are defined in the C standard header file `<limits.h>`. The C++ Standard Library header `<limits>` includes `<climits>`, which includes `<limits.h>`.

Microsoft C also permits the declaration of sized integer variables, which are integral types of size 8-, 16-, 32- or 64-bits. For more information on sized integers in C, see Sized Integer Types.

Limits on Integer Constants

CONSTANT	MEANING	VALUE
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type `signed char`.	-128
SCHAR_MAX	Maximum value for a variable of type `signed char`.	127
UCHAR_MAX	Maximum value for a variable of type `unsigned char`.	255 (0xff)
CHAR_MIN	Minimum value for a variable of type `char`.	-128; 0 if /J option used
CHAR_MAX	Maximum value for a variable of type `char`.	127; 255 if /J option used
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	5
SHRT_MIN	Minimum value for a variable of type `short`.	-32768
SHRT_MAX	Maximum value for a variable of type `short`.	32767
USHRT_MAX	Maximum value for a variable of type `unsigned short`.	65535 (0xffff)
INT_MIN	Minimum value for a variable of type `int`.	-2147483647 - 1
INT_MAX	Maximum value for a variable of type `int`.	2147483647

CONSTANT	MEANING	VALUE
UINT_MAX	Maximum value for a variable of type `unsigned int`.	4294967295 (0xffffffff)
LONG_MIN	Minimum value for a variable of type `long`.	-2147483647 - 1
LONG_MAX	Maximum value for a variable of type `long`.	2147483647
ULONG_MAX	Maximum value for a variable of type `unsigned long`.	4294967295 (0xffffffff)
LLONG_MIN	Minimum value for a variable of type `long long`.	-9,223,372,036,854,775,807 - 1
LLONG_MAX	Maximum value for a variable of type `long long`.	9,223,372,036,854,775,807
ULLONG_MAX	Maximum value for a variable of type `unsigned long long`.	18,446,744,073,709,551,615 (0xffffffffffffffff)

If a value exceeds the largest integer representation, the Microsoft compiler generates an error.

END Microsoft Specific

See also

C Integer Constants

C Character Constants

12/22/2021 • 2 minutes to read • Edit Online

A "character constant" is formed by enclosing a single character from the representable character set within single quotation marks (' '). Character constants are used to represent characters in the execution character set.

Syntax

character-constant: **'** *c-char-sequence* **'**

L' *c-char-sequence* **'**

c-char-sequence: *c-char*

c-char-sequence c-char

c-char: Any member of the source character set except the single quotation mark ('), backslash (\), or newline character

escape-sequence

escape-sequence: *simple-escape-sequence*

octal-escape-sequence

hexadecimal-escape-sequence

simple-escape-sequence: one of **\a \b \f \n \r \t \v**

\' \" \\ \?

octal-escape-sequence: **** *octal-digit*

**** *octal-digit octal-digit*

**** *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence: **\x** *hexadecimal-digit*

hexadecimal-escape-sequence hexadecimal-digit

See also

C Constants

Character Types

12/22/2021 • 2 minutes to read • Edit Online

An integer character constant not preceded by the letter **L** has type `int`. The value of an integer character constant containing a single character is the numerical value of the character interpreted as an integer. For example, the numerical value of the character `a` is 97 in decimal and 61 in hexadecimal.

Syntactically, a "wide-character constant" is a character constant prefixed by the letter **L**. A wide-character constant has type `wchar_t`, an integer type defined in the STDDEF.H header file. For example:

```
char    schar =  'x';   /* A character constant          */
wchar_t wchar = L'x';   /* A wide-character constant for
                           the same character           */
```

Wide-character constants are 16 bits wide and specify members of the extended execution character set. They allow you to express characters in alphabets that are too large to be represented by type `char`. See Multibyte and Wide Characters for more information about wide characters.

See also

C Character Constants

Execution Character Set

12/22/2021 • 2 minutes to read • Edit Online

This content often refers to the "execution character set." The execution character set is not necessarily the same as the source character set used for writing C programs. The execution character set includes all characters in the source character set as well as the null character, newline character, backspace, horizontal tab, vertical tab, carriage return, and escape sequences. The source and execution character sets may differ in other implementations.

See also

C Character Constants

Escape Sequences

12/22/2021 • 2 minutes to read • Edit Online

Character combinations consisting of a backslash (\) followed by a letter or by a combination of digits are called "escape sequences." To represent a newline character, single quotation mark, or certain other characters in a character constant, you must use escape sequences. An escape sequence is regarded as a single character and is therefore valid as a character constant.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers. They are also used to provide literal representations of nonprinting characters and characters that usually have special meanings, such as the double quotation mark ("). The following table lists the ANSI escape sequences and what they represent.

Note that the question mark preceded by a backslash (**\?**) specifies a literal question mark in cases where the character sequence would be misinterpreted as a trigraph. See Trigraphs for more information.

Escape Sequences

ESCAPE SEQUENCE	REPRESENTS
\a	Bell (alert)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
****	Backslash
\?	Literal question mark
**** *ooo*	ASCII character in octal notation
\x *hh*	ASCII character in hexadecimal notation

ESCAPE SEQUENCE	REPRESENTS
\x *hhhh*	Unicode character in hexadecimal notation if this escape sequence is used in a wide-character constant or a Unicode string literal. For example, `WCHAR f = L'\x4e00'` or `WCHAR b[] = L"The Chinese character for one is \x4e00"` .

Microsoft Specific

If a backslash precedes a character that does not appear in the table, the compiler handles the undefined character as the character itself. For example, `\c` is treated as an `c`.

END Microsoft Specific

Escape sequences allow you to send nongraphic control characters to a display device. For example, the ESC character (**\033**) is often used as the first character of a control command for a terminal or printer. Some escape sequences are device-specific. For instance, the vertical tab and form feed escape sequences (**\v** and **\f**) do not affect screen output, but they do perform appropriate printer operations.

You can also use the backslash (****) as a continuation character. When a newline character (equivalent to pressing the RETURN key) immediately follows the backslash, the compiler ignores the backslash and the newline character and treats the next line as part of the previous line. This is useful primarily for preprocessor definitions longer than a single line. For example:

```
#define assert(exp) \
( (exp) ? (void) 0:_assert( #exp, __FILE__, __LINE__ ) )
```

See also

C Character Constants

Octal and Hexadecimal Character Specifications

12/22/2021 • 2 minutes to read • Edit Online

The sequence *ooo* means you can specify any character in the ASCII character set as a three-digit octal character code. The numerical value of the octal integer specifies the value of the desired character or wide character.

Similarly, the sequence **\x***hhh* allows you to specify any ASCII character as a hexadecimal character code. For example, you can give the ASCII backspace character as the normal C escape sequence (**\b**), or you can code it as **\010** (octal) or **\x008** (hexadecimal).

You can use only the digits 0 through 7 in an octal escape sequence. Octal escape sequences can never be longer than three digits and are terminated by the first character that is not an octal digit. Although you do not need to use all three digits, you must use at least one. For example, the octal representation is **\10** for the ASCII backspace character and **\101** for the letter A, as given in an ASCII chart.

Similarly, you must use at least one digit for a hexadecimal escape sequence, but you can omit the second and third digits. Therefore you could specify the hexadecimal escape sequence for the backspace character as either **\x8**, **\x08**, or **\x008**.

The value of the octal or hexadecimal escape sequence must be in the range of representable values for type `unsigned char` for a character constant and type `wchar_t` for a wide-character constant. See Multibyte and Wide Characters for information on wide-character constants.

Unlike octal escape constants, the number of hexadecimal digits in an escape sequence is unlimited. A hexadecimal escape sequence terminates at the first character that is not a hexadecimal digit. Because hexadecimal digits include the letters **a** through **f**, care must be exercised to make sure the escape sequence terminates at the intended digit. To avoid confusion, you can place octal or hexadecimal character definitions in a macro definition:

```
#define Bell '\x07'
```

For hexadecimal values, you can break the string to show the correct value clearly:

```
"\xabc"    /* one character  */
"\xab" "c" /* two characters */
```

See also

C Character Constants

C String Literals

12/22/2021 • 2 minutes to read • Edit Online

A "string literal" is a sequence of characters from the source character set enclosed in double quotation marks (" "). String literals are used to represent a sequence of characters which, taken together, form a null-terminated string. You must always prefix wide-string literals with the letter **L**.

Syntax

string-literal:
  " *s-char-sequence*$_{\text{opt}}$ **"**
  L" *s-char-sequence*$_{\text{opt}}$ **"**

s-char-sequence:
  s-char

  s-char-sequence s-char

s-char:
  any member of the source character set except the double quotation mark ("), backslash (\), or newline character

  escape-sequence

Remarks

The example below is a simple string literal:

```
char *amessage = "This is a string literal.";
```

All escape codes listed in the Escape Sequences table are valid in string literals. To represent a double quotation mark in a string literal, use the escape sequence **\"**. The single quotation mark (**'**) can be represented without an escape sequence. The backslash (****) must be followed with a second backslash (****) when it appears within a string. When a backslash appears at the end of a line, it is always interpreted as a line-continuation character.

See also

Elements of C

Type for String Literals

12/22/2021 • 2 minutes to read • Edit Online

String literals have type array of `char` (that is, **char[ ]**). (Wide-character strings have type array of `wchar_t` (that is, **wchar_t[ ]**).) This means that a string is an array with elements of type `char`. The number of elements in the array is equal to the number of characters in the string plus one for the terminating null character.

See also

C String Literals

Storage of String Literals

12/22/2021 • 2 minutes to read • Edit Online

The characters of a literal string are stored in order at contiguous memory locations. An escape sequence (such as `\\` or `\"`) within a string literal counts as a single character. A null character (represented by the `\0` escape sequence) is automatically appended to, and marks the end of, each string literal. (This occurs during translation phase 7.) Note that the compiler may not store two identical strings at two different addresses. /GF forces the compiler to place a single copy of identical strings into the executable file.

Remarks

Microsoft Specific

Strings have static storage duration. See Storage Classes for information about storage duration.

END Microsoft Specific

See also

C String Literals

String Literal Concatenation

12/22/2021 • 2 minutes to read • Edit Online

To form string literals that take up more than one line, you can concatenate the two strings. To do this, type a backslash, then press the RETURN key. The backslash causes the compiler to ignore the following newline character. For example, the string literal

```
"Long strings can be bro\
ken into two or more pieces."
```

is identical to the string

```
"Long strings can be broken into two or more pieces."
```

String concatenation can be used anywhere you might previously have used a backslash followed by a newline character to enter strings longer than one line.

To force a new line within a string literal, enter the newline escape sequence (**\n**) at the point in the string where you want the line broken, as follows:

```
"Enter a number between 1 and 100\nOr press Return"
```

Because strings can start in any column of the source code and long strings can be continued in any column of a succeeding line, you can position strings to enhance source-code readability. In either case, their on-screen representation when output is unaffected. For example:

```
printf_s ( "This is the first half of the string, "
           "this is the second half ") ;
```

As long as each part of the string is enclosed in double quotation marks, the parts are concatenated and output as a single string. This concatenation occurs according to the sequence of events during compilation specified by translation phases.

```
"This is the first half of the string, this is the second half"
```

A string pointer, initialized as two distinct string literals separated only by white space, is stored as a single string (pointers are discussed in Pointer Declarations). When properly referenced, as in the following example, the result is identical to the previous example:

```
char *string = "This is the first half of the string, "
               "this is the second half";

printf_s( "%s" , string ) ;
```

In translation phase 6, the multibyte-character sequences specified by any sequence of adjacent string literals or adjacent wide-string literals are concatenated into a single multibyte-character sequence. Therefore, do not design programs to allow modification of string literals during execution. The ANSI C standard specifies that the result of modifying a string is undefined.

See also

C String Literals

Maximum String Length

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

ANSI compatibility requires a compiler to accept up to 509 characters in a string literal after concatenation. The maximum length of a string literal allowed in Microsoft C is approximately 2,048 bytes. However, if the string literal consists of parts enclosed in double quotation marks, the preprocessor concatenates the parts into a single string, and for each line concatenated, it adds an extra byte to the total number of bytes.

For example, suppose a string consists of 40 lines with 50 characters per line (2,000 characters), and one line with 7 characters, and each line is surrounded by double quotation marks. This adds up to 2,007 bytes plus one byte for the terminating null character, for a total of 2,008 bytes. On concatenation, an extra character is added for each of the first 40 lines. This makes a total of 2,048 bytes. Note, however, that if line continuations (\) are used instead of double quotation marks, the preprocessor does not add an extra character for each line.

While an individual quoted string cannot be longer than 2048 bytes, a string literal of roughly 65535 bytes can be constructed by concatenating strings.

END Microsoft Specific

See also

C String Literals

Punctuation and Special Characters

12/22/2021 • 2 minutes to read • Edit Online

The punctuation and special characters in the C character set have various uses, from organizing program text to defining the tasks that the compiler or the compiled program carries out. They do not specify an operation to be performed. Some punctuation symbols are also operators (see Operators). The compiler determines their use from context.

Syntax

`punctuator` : one of ( ) [ ] { } * , : = ; ... #

These characters have special meanings in C. Their uses are described throughout this book. The pound sign (**#**) can occur only in preprocessing directives.

See also

Elements of C

Program Structure

12/22/2021 • 2 minutes to read • Edit Online

This section gives an overview of C programs and program execution. Terms and features important to understanding C programs and components are also introduced. Topics discussed include:

- Source files and source programs
- The main function and program execution
- Parsing command-line arguments
- Lifetime, scope, visibility, and linkage
- Name spaces

Because this section is an overview, the topics discussed contain introductory material only. See the cross-referenced information for more detailed explanations.

See also

C Language Reference

Source Files and Source Programs

12/22/2021 • 2 minutes to read • Edit Online

A source program can be divided into one or more "source files," or "translation units." The input to the compiler is called a "translation unit."

Syntax

translation-unit:
   external-declaration
   translation-unit external-declaration

external-declaration:
   function-definition
   declaration

Overview of Declarations gives the syntax for the `declaration` nonterminal, and the *Preprocessor Reference* explains how the translation unit is processed.

> **NOTE**
>
> See the introduction to C Language Syntax Summary, for an explanation of the ANSI syntax conventions.

The components of a translation unit are external declarations that include function definitions and identifier declarations. These declarations and definitions can be in source files, header files, libraries, and other files the program needs. You must compile each translation unit and link the resulting object files to make a program.

A C "source program" is a collection of directives, pragmas, declarations, definitions, statement blocks, and functions. To be valid components of a Microsoft C program, each must have the syntax described in this book, although they can appear in any order in the program (subject to the rules outlined throughout this book). However, the location of these components in a program does affect how variables and functions can be used in a program. (See Lifetime, Scope, Visibility, and Linkage for more information.)

Source files need not contain executable statements. For example, you may find it useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique makes the definitions easy to find and update when necessary. For the same reason, constants and macros are often organized into separate files called "include files" or "header files" that can be referenced in source files as required. See the *Preprocessor Reference* for information about macros and include files.

See also

Program Structure

Directives to the Preprocessor

12/22/2021 • 2 minutes to read • Edit Online

A "directive" instructs the C preprocessor to perform a specific action on the text of the program before compilation. Preprocessor directives are fully described in the *Preprocessor Reference*. This example uses the preprocessor directive `#define`:

```
#define MAX 100
```

This statement tells the compiler to replace each occurrence of `MAX` by `100` before compilation. The C compiler preprocessor directives are:

#DEFINE	#ENDIF	#IFDEF	#LINE
`#elif`	`#error`	**#ifndef**	**#pragma**
`#else`	`#if`	`#include`	`#undef`

See also

Source Files and Source Programs

C Pragmas

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

A *pragma* instructs the compiler to perform a particular action at compile time. Pragmas vary from compiler to compiler. For example, you can use the `optimize` pragma to set the optimizations to perform on your program. The Microsoft C pragmas are:

`alloc_text`
`auto_inline`
`bss_seg`
`check_stack`
`code_seg`
`comment`
`component`
`const_seg`
`data_seg`
`deprecated`

`detect_mismatch`
`fenv_access`
`float_control`
`fp_contract`
`function`
`hdrstop`
`include_alias`
`inline_depth`
`inline_recursion`

`intrinsic`
`make_public`
`managed`
`message`
`omp`
`once`
`optimize`
`pack`
`pop_macro`

`push_macro`
`region`, `endregion`
`runtime_checks`
`section`
`setlocale`
`strict_gs_check`
`system_header`
`unmanaged`
`warning`

See Pragma Directives and the `__Pragma` Keyword for a description of the Microsoft C compiler pragmas.

END Microsoft Specific

See also

Source Files and Source Programs

C Declarations and Definitions

12/22/2021 • 2 minutes to read • Edit Online

A "declaration" establishes an association between a particular variable, function, or type and its attributes. Overview of Declarations gives the ANSI syntax for the `declaration` nonterminal. A declaration also specifies where and when an identifier can be accessed (the "linkage" of an identifier). See Lifetime, Scope, Visibility, and Linkage for information about linkage.

A "definition" of a variable establishes the same associations as a declaration but also causes storage to be allocated for the variable.

For example, the `main`, `find`, and `count` functions and the `var` and `val` variables are defined in one source file, in this order:

```
int main() {}

int var = 0;
double val[MAXVAL];
char find( fileptr ) {}
int count( double f ) {}
```

The variables `var` and `val` can be used in the `find` and `count` functions; no further declarations are needed. But these names are not visible (cannot be accessed) in `main`.

See also

Source Files and Source Programs

Function Declarations and Definitions

12/22/2021 • 2 minutes to read • Edit Online

Function prototypes establish the name of the function, its return type, and the type and number of its formal parameters. A function definition includes the function body.

Remarks

Both function and variable declarations can appear inside or outside a function definition. Any declaration within a function definition is said to appear at the "internal" or "local" level. A declaration outside all function definitions is said to appear at the "external," "global," or "file scope" level. Variable definitions, like declarations, can appear at the internal level (within a function definition) or at the external level (outside all function definitions). Function definitions always occur at the external level. Function definitions are discussed further in Function Definitions. Function prototypes are covered in Function Prototypes.

See also

Source Files and Source Programs

Blocks

12/22/2021 • 2 minutes to read • Edit Online

A sequence of declarations, definitions, and statements enclosed within curly braces (**{ }**) is called a "block." There are two types of blocks in C. The "compound statement," a statement composed of one or more statements (see The Compound Statement), is one type of block. The other, the "function definition," consists of a compound statement (the body of the function) plus the function's associated "header" (the function name, return type, and formal parameters). A block within other blocks is said to be "nested."

Note that while all compound statements are enclosed within curly braces, not everything enclosed within curly braces constitutes a compound statement. For example, although the specifications of array, structure, or enumeration elements can appear within curly braces, they are not compound statements.

See also

Source Files and Source Programs

Example Program

12/22/2021 • 2 minutes to read • Edit Online

The following C source program consists of two source files. It gives an overview of some of the various declarations and definitions possible in a C program. Later sections in this book describe how to write these declarations, definitions, and initializations, and how to use C keywords such as `static` and `extern`. The `printf` function is declared in the C header file STDIO.H.

The `main` and `max` functions are assumed to be in separate files, and execution of the program begins with the `main` function. No explicit user functions are executed before `main`.

```
/*********************************************************************
                    FILE1.C - main function
*********************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3
#include <stdio.h>

int a = 1;                       // Defining declarations
int b = 2;                       //  of external variables

extern int max( int a, int b );  // Function prototype

int main()                       // Function definition
{                                //  for main function
    int c;                       // Definitions for
    int d;                       //  two uninitialized
                                 //  local variables

    extern int u;                // Referencing declaration
                                 //  of external variable
                                 //  defined elsewhere
    static int v;                // Definition of variable
                                 //  with continuous lifetime

    int w = ONE, x = TWO, y = THREE;
    int z = 0;

    z = max( x, y );             // Executable statements
    w = max( z, w );
    printf_s( "%d %d\n", z, w );
    return 0;
}

/********************************************************************
            FILE2.C - definition of max function
********************************************************************/

int max( int a, int b )          // Note formal parameters are
                                 //  included in function header
{
    if( a > b )
        return( a );
    else
        return( b );
}
```

FILE1.C contains the prototype for the `max` function. This kind of declaration is sometimes called a "forward declaration" because the function is declared before it is used. The definition for the `main` function includes calls to `max`.

The lines beginning with `#define` are preprocessor directives. These directives tell the preprocessor to replace the identifiers `ONE`, `TWO`, and `THREE` with the numbers `1`, `2`, and `3`, respectively, throughout FILE1.C. However, the directives do not apply to FILE2.C, which is compiled separately and then linked with FILE1.C. The line beginning with `#include` tells the compiler to include the file STDIO.H, which contains the prototype for the `printf` function. Preprocessor directives are explained in the *Preprocessor Reference*.

FILE1.C uses defining declarations to initialize the global variables `a` and `b`. The local variables `c` and `d` are declared but not initialized. Storage is allocated for all these variables. The static and external variables, `u` and `v`, are automatically initialized to 0. Therefore only `a`, `b`, `u`, and `v` contain meaningful values when declared because they are initialized, either explicitly or implicitly. FILE2.C contains the function definition for `max`. This definition satisfies the calls to `max` in FILE1.C.

The lifetime and visibility of identifiers are discussed in Lifetime, Scope, Visibility, and Linkage. For more information on functions, see Functions.

See also

Source Files and Source Programs

main Function and Program Execution

12/22/2021 • 2 minutes to read • Edit Online

Every C program has a primary (main) function that must be named **main**. If your code adheres to the Unicode programming model, you can use the wide-character version of **main**, **wmain**. The **main** function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of **main**, although it can terminate at other points in the program for a variety of reasons. At times, perhaps when a certain error is detected, you may want to force the termination of a program. To do so, use the **exit** function. See the *Run-Time Library Reference* for information on and an example using the exit function.

Syntax

```
main( int argc, char *argv[ ], char *envp[ ] )
```

Remarks

Functions within the source program perform one or more specific tasks. The **main** function can call these functions to perform their respective tasks. When **main** calls another function, it passes execution control to the function, so that execution begins at the first statement in the function. A function returns control to **main** when a `return` statement is executed or when the end of the function is reached.

You can declare any function, including **main**, to have parameters. The term "parameter" or "formal parameter" refers to the identifier that receives a value passed to a function. See Parameters for information on passing arguments to parameters. When one function calls another, the called function receives values for its parameters from the calling function. These values are called "arguments." You can declare formal parameters to **main** so that it can receive arguments from the command line using this format:

When you want to pass information to the **main** function, the parameters are traditionally named `argc` and `argv`, although the C compiler does not require these names. The types for `argc` and `argv` are defined by the C language. Traditionally, if a third parameter is passed to **main**, that parameter is named `envp`. Examples later in this section show how to use these three parameters to access command-line arguments. The following sections explain these parameters.

See Using wmain for a description of the wide-character version of **main**.

See also

main function and command-line arguments (C++)
Parsing C Command-Line Arguments

Using wmain

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

In the Unicode programming model, you can define a wide-character version of the **main** function. Use **wmain** instead of **main** if you want to write portable code that adheres to the Unicode programming model.

Syntax

```
wmain( int argc, wchar_t *argv[ ], wchar_t *envp[ ] )
```

Remarks

You declare formal parameters to **wmain** using a similar format to **main**. You can then pass wide-character arguments and, optionally, a wide-character environment pointer to the program. The `argv` and `envp` parameters to **wmain** are of type `wchar_t*`. For example:

If your program uses a **main** function, the multibyte-character environment is created by the run-time library at program startup. A wide-character copy of the environment is created only when needed (for example, by a call to the `_wgetenv` or `_wputenv` functions). On the first call to `_wputenv`, or on the first call to `_wgetenv` if an MBCS environment already exists, a corresponding wide-character string environment is created and is then pointed to by the `_wenviron` global variable, which is a wide-character version of the `_environ` global variable. At this point, two copies of the environment (MBCS and Unicode) exist simultaneously and are maintained by the operating system throughout the life of the program.

Similarly, if your program uses a **wmain** function, a wide-character environment is created at program startup and is pointed to by the `_wenviron` global variable. An MBCS (ASCII) environment is created on the first call to `_putenv` or `getenv`, and is pointed to by the `_environ` global variable.

For more information on the MBCS environment, see Internationalization in the *Run-Time Library Reference.*

END Microsoft Specific

See also

main Function and Program Execution

Argument Description

12/22/2021 • 2 minutes to read • Edit Online

The `argc` parameter in the **main** and **wmain** functions is an integer specifying how many arguments are passed to the program from the command line. Since the program name is considered an argument, the value of `argc` is at least one.

Remarks

The `argv` parameter is an array of pointers to null-terminated strings representing the program arguments. Each element of the array points to a string representation of an argument passed to **main** (or **wmain**). (For information about arrays, see Array Declarations.) The `argv` parameter can be declared either as an array of pointers to type `char` (`char *argv[]`) or as a pointer to pointers to type `char` (`char **argv`). For **wmain**, the `argv` parameter can be declared either as an array of pointers to type `wchar_t` (`wchar_t *argv[]`) or as a pointer to pointers to type `wchar_t` (`wchar_t **argv`).

By convention, `argv` **[0]** is the command with which the program is invoked. However, it is possible to spawn a process using CreateProcess and if you use both the first and second arguments (`lpApplicationName` and `lpCommandLine`), `argv` **[0]** may not be the executable name; use GetModuleFileName to retrieve the executable name.

The last pointer (`argv[argc]`) is **NULL**. (See getenv in the *Run-Time Library Reference* for an alternative method for getting environment variable information.)

Microsoft Specific

The `envp` parameter is a pointer to an array of null-terminated strings that represent the values set in the user's environment variables. The `envp` parameter can be declared as an array of pointers to `char` (`char *envp[]`) or as a pointer to pointers to `char` (`char **envp`). In a **wmain** function, the `envp` parameter can be declared as an array of pointers to `wchar_t` (`wchar_t *envp[]`) or as a pointer to pointers to `wchar_t` (`wchar_t **envp`). The end of the array is indicated by a **NULL** *pointer. Note that the environment block passed to **main** or **wmain** is a "frozen" copy of the current environment. If you subsequently change the environment via a call to **_putenv** or `_wputenv` , the current environment (as returned by `getenv` / `_wgetenv` and the `_environ` or `_wenviron` variables) will change, but the block pointed to by `envp` will not change. The `envp` parameter is ANSI compatible in C, but not in C++.

END Microsoft Specific

See also

main Function and Program Execution

Expanding wildcard arguments

12/22/2021 • 2 minutes to read • Edit Online

Wildcard argument expansion is Microsoft-specific.

When you run a C program, you can use either of the two wildcards, the question mark (`?`) and the asterisk (`*`), to specify filename and path arguments on the command line.

By default, wildcards aren't expanded in command-line arguments. You can replace the normal argument vector `argv` loading routine with a version that does expand wildcards by linking with the *`setargv.obj`* or *`wsetargv.obj`* file. If your program uses a `main` function, link with *`setargv.obj`*. If your program uses a `wmain` function, link with *`wsetargv.obj`*. Both of these have equivalent behavior.

To link with *`setargv.obj`* or *`wsetargv.obj`*, use the `/link` option. For example:

```
cl example.c /link setargv.obj
```

The wildcards are expanded in the same manner as operating system commands.

See also

Link options
`main` function and program execution

Parsing C command-line arguments

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

Microsoft C startup code uses the following rules when interpreting arguments given on the operating system command line:

- Arguments are delimited by whitespace characters, which are either spaces or tabs.
- The first argument (`argv[0]`) is treated specially. It represents the program name. Because it must be a valid pathname, parts surrounded by double quote marks (`"`) are allowed. The double quote marks aren't included in the `argv[0]` output. The parts surrounded by double quote marks prevent interpretation of a space or tab character as the end of the argument. The later rules in this list don't apply.
- A string surrounded by double quote marks is interpreted as a single argument, whether it contains whitespace characters or not. A quoted string can be embedded in an argument. The caret (`^`) isn't recognized as an escape character or delimiter. Within a quoted string, a pair of double quote marks is interpreted as a single escaped double quote mark. If the command line ends before a closing double quote mark is found, then all the characters read so far are output as the last argument.
- A double quote mark preceded by a backslash (`\"`) is interpreted as a literal double quote mark (`"`).
- Backslashes are interpreted literally, unless they immediately precede a double quote mark.
- If an even number of backslashes is followed by a double quote mark, then one backslash (`\`) is placed in the `argv` array for every pair of backslashes (`\\`), and the double quote mark (`"`) is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quote mark, then one backslash (`\`) is placed in the `argv` array for every pair of backslashes (`\\`). The double quote mark is interpreted as an escape sequence by the remaining backslash, causing a literal double quote mark (`"`) to be placed in `argv`.

This list illustrates the rules above by showing the interpreted result passed to `argv` for several examples of command-line arguments. The output listed in the second, third, and fourth columns is from the ARGS.C program that follows the list.

COMMAND-LINE INPUT	`ARGV[1]`	`ARGV[2]`	`ARGV[3]`
`"a b c" d e`	`a b c`	`d`	`e`
`"ab\"c" "\\" d`	`ab"c`	`\`	`d`
`a\\\b d"e f"g h`	`a\\\b`	`de fg`	`h`
`a\\\"b c d`	`a\"b`	`c`	`d`
`a\\\\"b c" d e`	`a\\b c`	`d`	`e`
`a"b"" c d`	`ab" c d`		

Example

Code

```
// ARGS.C illustrates the following variables used for accessing
// command-line arguments and environment variables:
// argc  argv  envp
//

#include <stdio.h>

int main( int argc, // Number of strings in array argv
char *argv[],      // Array of command-line argument strings
char **envp )      // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    printf_s( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf_s( "  argv[%d]   %s\n", count, argv[count] );

    // Display each environment variable.
    printf_s( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf_s( "  %s\n", *(envp++) );

    return;
}
```

One example of output from this program is:

```
Command-line arguments:
  argv[0]   C:\MSC\ARGS.EXE

Environment variables:
  COMSPEC=C:\NT\SYSTEM32\CMD.EXE
  PATH=c:\nt;c:\binb;c:\binr;c:\nt\system32;c:\word;c:\help;c:\msc;c:\;
  PROMPT=[$p]
  TEMP=c:\tmp
  TMP=c:\tmp
  EDITORS=c:\binr
  WINDIR=c:\nt
```

END Microsoft Specific

See also

`main` function and program execution

Customizing C command-line processing

12/22/2021 • 2 minutes to read • Edit Online

If your program doesn't take command-line arguments, you can suppress the command-line processing routine to save a small amount of space. To suppress its use, include the `noarg.obj` file (for both `main` and `wmain`) in your `/link` compiler options or your `LINK` command line.

Similarly, if you never access the environment table through the `envp` argument, you can suppress the internal environment-processing routine. To suppress its use, include the `noenv.obj` file (for both `main` and `wmain`) in your `/link` compiler options or your `LINK` command line.

For more information on runtime startup linker options, see Link options.

Your program might make calls to the `spawn` or `exec` family of routines in the C runtime library. If it does, you shouldn't suppress the environment-processing routine, since it's used to pass an environment from the parent process to the child process.

See also

`main` function and program execution
Link options.

Lifetime, Scope, Visibility, and Linkage

12/22/2021 • 2 minutes to read • Edit Online

To understand how a C program works, you must understand the rules that determine how variables and functions can be used in the program. Several concepts are crucial to understanding these rules:

- Lifetime
- Scope and visibility
- Linkage

See also

Program Structure

Lifetime

12/22/2021 • 2 minutes to read • Edit Online

"Lifetime" is the period during execution of a program in which a variable or function exists. The storage duration of the identifier determines its lifetime.

An identifier declared with the *storage-class-specifier* `static` has static storage duration. Identifiers with static storage duration (also called "global") have storage and a defined value for the duration of a program. Storage is reserved and the identifier's stored value is initialized only once, before program startup. An identifier declared with external or internal linkage also has static storage duration (see Linkage).

An identifier declared without the `static` storage-class specifier has automatic storage duration if it is declared inside a function. An identifier with automatic storage duration (a "local identifier") has storage and a defined value only within the block where the identifier is defined or declared. An automatic identifier is allocated new storage each time the program enters that block, and it loses its storage (and its value) when the program exits the block. Identifiers declared in a function with no linkage also have automatic storage duration.

The following rules specify whether an identifier has global (static) or local (automatic) lifetime:

- All functions have static lifetime. Therefore they exist at all times during program execution. Identifiers declared at the external level (that is, outside all blocks in the program at the same level of function definitions) always have global (static) lifetimes.
- If a local variable has an initializer, the variable is initialized each time it is created (unless it is declared as `static`). Function parameters also have local lifetime. You can specify global lifetime for an identifier within a block by including the `static` storage-class specifier in its declaration. Once declared `static`, the variable retains its value from one entry of the block to the next.

Although an identifier with a global lifetime exists throughout the execution of the source program (for example, an externally declared variable or a local variable declared with the `static` keyword), it may not be visible in all parts of the program. See Scope and Visibility for information about visibility, and see Storage Classes for a discussion of the *storage-class-specifier* nonterminal.

Memory can be allocated as needed (dynamic) if created through the use of special library routines such as `malloc`. Since dynamic memory allocation uses library routines, it is not considered part of the language. See the malloc function in the *Run-Time Library Reference*.

See also

Lifetime, Scope, Visibility, and Linkage

Scope and Visibility

12/22/2021 • 2 minutes to read • Edit Online

An identifier's "visibility" determines the portions of the program in which it can be referenced — its "scope." An identifier is visible (i.e., can be used) only in portions of a program encompassed by its "scope," which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears. The scope of an identifier is the part of the program in which the name can be used. This is sometimes called "lexical scope." There are four kinds of scope: function, file, block, and function prototype.

All identifiers except labels have their scope determined by the level at which the declaration occurs. The following rules for each kind of scope govern the visibility of identifiers within a program:

File scope The declarator or type specifier for an identifier with file scope appears outside any block or list of parameters and is accessible from any place in the translation unit after its declaration. Identifier names with file scope are often called "global" or "external." The scope of a global identifier begins at the point of its definition or declaration and terminates at the end of the translation unit.

Function scope A label is the only kind of identifier that has function scope. A label is declared implicitly by its use in a statement. Label names must be unique within a function. (For more information about labels and label names, see The goto and Labeled Statements.)

Block scope The declarator or type specifier for an identifier with block scope appears inside a block or within the list of formal parameter declarations in a function definition. It is visible only from the point of its declaration or definition to the end of the block containing its declaration or definition. Its scope is limited to that block and to any blocks nested in that block and ends at the curly brace that closes the associated block. Such identifiers are sometimes called "local variables."

Function-prototype scope The declarator or type specifier for an identifier with function-prototype scope appears within the list of parameter declarations in a function prototype (not part of the function declaration). Its scope terminates at the end of the function declarator.

The appropriate declarations for making variables visible in other source files are described in Storage Classes. However, variables and functions declared at the external level with the `static` storage-class specifier are visible only within the source file in which they are defined. All other functions are globally visible.

See also

Lifetime, Scope, Visibility, and Linkage

Summary of Lifetime and Visibility

12/22/2021 • 2 minutes to read • Edit Online

The following table is a summary of lifetime and visibility characteristics for most identifiers. The first three columns give the attributes that define lifetime and visibility. An identifier with the attributes given by the first three columns has the lifetime and visibility shown in the fourth and fifth columns. However, the table does not cover all possible cases. Refer to Storage Classes for more information.

Summary of Lifetime and Visibility

ATTRIBUTES:		STORAGE-CLASS	RESULT:	
LEVEL	ITEM	SPECIFIER	LIFETIME	VISIBILITY
File scope	Variable definition	`static`	Global	Remainder of source file in which it occurs
	Variable declaration	`extern`	Global	Remainder of source file in which it occurs
	Function prototype or definition	`static`	Global	Single source file
	Function prototype	`extern`	Global	Remainder of source file
Block scope	Variable declaration	`extern`	Global	Block
	Variable definition	`static`	Global	Block
	Variable definition	`auto` or `register`	Local	Block

Example

Description

The following example illustrates blocks, nesting, and visibility of variables:

Code

```
// Lifetime_and_Visibility.c

#include <stdio.h>

int i = 1;  // i defined at external level

int main()  // main function defined at external level
{
    printf_s( "%d\n", i ); // Prints 1 (value of external level i)
    {                                 // Begin first nested block
        int i = 2, j = 3;          // i and j defined at internal level
        printf_s( "%d %d\n", i, j );  // Prints 2, 3
        {                             // Begin second nested block
            int i = 0;                // i is redefined
            printf_s( "%d %d\n", i, j ); // Prints 0, 3
        }                             // End of second nested block
        printf_s( "%d\n", i );        // Prints 2 (outer definition
                                      //  restored)
    }                                 // End of first nested block
    printf_s( "%d\n", i );            // Prints 1 (external level
                                      // definition restored)
    return 0;
}
```

Comments

In this example, there are four levels of visibility: the external level and three block levels. The values are printed to the screen as noted in the comments following each statement.

See also

Lifetime, Scope, Visibility, and Linkage

Linkage

12/22/2021 • 2 minutes to read • Edit Online

Identifier names can refer to different identifiers in different scopes. An identifier declared in different scopes or in the same scope more than once can be made to refer to the same identifier or function by a process called "linkage." Linkage determines the portions of the program in which an identifier can be referenced (its "visibility"). There are three kinds of linkage: internal, external, and no linkage.

See also

Using extern to Specify Linkage

Internal Linkage

12/22/2021 • 2 minutes to read • Edit Online

If the declaration of a file-scope identifier for an object or a function contains the *storage-class-specifier* `static`, the identifier has internal linkage. Otherwise, the identifier has external linkage. See Storage Classes for a discussion of the *storage-class-specifier* nonterminal.

Within one translation unit, each instance of an identifier with internal linkage denotes the same identifier or function. Internally linked identifiers are unique to a translation unit.

See also

Using extern to Specify Linkage

External Linkage

12/22/2021 • 2 minutes to read • Edit Online

If the first declaration at file-scope level for an identifier does not use the `static` storage-class specifier, the object has external linkage.

If the declaration of an identifier for a function has no *storage-class-specifier*, its linkage is determined exactly as if it were declared with the *storage-class-specifier* `extern`. If the declaration of an identifier for an object has file scope and no *storage-class-specifier*, its linkage is external.

An identifier's name with external linkage designates the same function or data object as does any other declaration for the same name with external linkage. The two declarations can be in the same translation unit or in different translation units. If the object or function also has global lifetime, the object or function is shared by the entire program.

See also

Using extern to Specify Linkage

No Linkage

12/22/2021 • 2 minutes to read • Edit Online

If a declaration for an identifier within a block does not include the `extern` storage-class specifier, the identifier has no linkage and is unique to the function.

The following identifiers have no linkage:

- An identifier declared to be anything other than an object or a function
- An identifier declared to be a function parameter
- A block-scope identifier for an object declared without the `extern` storage-class specifier

If an identifier has no linkage, declaring the same name again (in a declarator or type specifier) in the same scope level generates a symbol redefinition error.

See also

Using extern to Specify Linkage

Name Spaces

12/22/2021 • 2 minutes to read • Edit Online

The compiler sets up "name spaces" to distinguish between the identifiers used for different kinds of items. The names within each name space must be unique to avoid conflict, but an identical name can appear in more than one name space. This means that you can use the same identifier for two or more different items, provided that the items are in different name spaces. The compiler can resolve references based on the syntactic context of the identifier in the program.

NOTE

Do not confuse the limited C notion of a name space with the C++ "namespace" feature. See Namespaces in the C++ Language Reference for more information.

This list describes the name spaces used in C.

Statement labels Named statement labels are part of statements. Definitions of statement labels are always followed by a colon but are not part of `case` labels. Uses of statement labels always immediately follow the keyword `goto`. Statement labels do not have to be distinct from other names or from label names in other functions.

Structure, union, and enumeration tags These tags are part of structure, union, and enumeration type specifiers and, if present, always immediately follow the reserved words `struct`, `union`, or `enum`. The tag names must be distinct from all other structure, enumeration, or union tags with the same visibility.

Members of structures or unions Member names are allocated in name spaces associated with each structure and union type. That is, the same identifier can be a component name in any number of structures or unions at the same time. Definitions of component names always occur within structure or union type specifiers. Uses of component names always immediately follow the member-selection operators (-> and .). The name of a member must be unique within the structure or union, but it does not have to be distinct from other names in the program, including the names of members of different structures and unions, or the name of the structure itself.

Ordinary identifiers All other names fall into a name space that includes variables, functions (including formal parameters and local variables), and enumeration constants. Identifier names have nested visibility, so you can redefine them within blocks.

Typedef names Typedef names cannot be used as identifiers in the same scope.

For example, since structure tags, structure members, and variable names are in three different name spaces, the three items named `student` in this example do not conflict. The context of each item allows correct interpretation of each occurrence of `student` in the program. (For information about structures, see Structure Declarations.)

```
struct student {
   char student[20];
   int class;
   int id;
   } student;
```

When `student` appears after the `struct` keyword, the compiler recognizes it as a structure tag. When `student` appears after a member-selection operator (`->` or `.`), the name refers to the structure member. In other contexts, `student` refers to the structure variable. However, overloading the tag name space is not recommended since it obscures meaning.

See also

Program Structure

Alignment (C11)

12/22/2021 • 3 minutes to read • Edit Online

One of the low-level features of C is the ability to specify the precise alignment of objects in memory to take maximum advantage of the hardware architecture.

CPUs read and write memory more efficiently when they store data at an address that's a multiple of the data size. For example, a 4-byte integer is accessed more efficiently if it's stored at an address that's a multiple of 4. When data isn't aligned, the CPU does more address calculation work to access the data.

By default, the compiler aligns data based on its size: `char` on a 1-byte boundary, `short` on a 2-byte boundary, `int`, `long`, and `float` on a 4-byte boundary, `double` on 8-byte boundary, and so on.

Additionally, by aligning frequently used data with the processor's cache line size, you can improve cache performance. For example, say you define a structure whose size is less than 32 bytes. You may want to use 32-byte alignment to ensure all instances of the structure are cached efficiently.

Usually, you don't need to worry about alignment. The compiler generally aligns data on natural boundaries that are based on the target processor and the size of the data. Data is aligned on up to 4-byte boundaries on 32-bit processors, and 8-byte boundaries on 64-bit processors. In some cases, however, you can achieve performance improvements, or memory savings, by specifying a custom alignment for your data structures.

Use the C11 keyword `_Alignof` to get the preferred alignment of a type or variable, and `_Alignas` to specify a custom alignment for a variable or user-defined type.

The convenience macros `alignof` and `alignas`, defined in `<stdalign.h>`, map directly to `_Alignof` and `_Alignas`, respectively. These macros match the keywords used in C++. So using the macros instead of the C keywords may be helpful for code portability if you share any code between the two languages.

`alignas` and `_Alignas` (C11)

Use `alignas` or `_Alignas` to specify custom alignment for a variable or user-defined type. They can be applied to a struct, union, enumeration, or variable.

`alignas` syntax

```
alignas(type)
alignas(constant-expression)
_Alignas(type)
_Alignas(constant-expression)
```

Remarks

`_Alignas` can't be used in the declaration of a typedef, bit-field, function, function parameter, or an object declared with the `register` specifier.

Specify an alignment that's a power of two, such as 1, 2, 4, 8, 16, and so on. Don't use a value smaller than the size of the type.

`struct` and `union` types have an alignment equal to the largest alignment of any member. Padding bytes are added within a `struct` to ensure individual member alignment requirements are met.

If there are several `alignas` specifiers in a declaration (for example, a `struct` with several members that have differing `alignas` specifiers), the alignment of the `struct` will be at least the value of the largest specifier.

`alignas` example

This example uses the convenience macro `alignof` because it's portable to C++. The behavior is the same if you use `_Alignof`.

```
// Compile with /std:c11

#include <stdio.h>
#include <stdalign.h>

typedef struct
{
    int value; // aligns on a 4-byte boundary. There will be 28 bytes of padding between value and alignas
    alignas(32) char alignedMemory[32]; // assuming a 32 byte friendly cache alignment
} cacheFriendly; // this struct will be 32-byte aligned because alignedMemory is 32-byte aligned and is the
largest alignment specified in the struct

int main()
{
    printf("sizeof(cacheFriendly): %d\n", sizeof(cacheFriendly)); // 4 bytes for int value + 32 bytes for
alignedMemory[] + padding to ensure  alignment
    printf("alignof(cacheFriendly): %d\n", alignof(cacheFriendly)); // 32 because alignedMemory[] is aligned
on a 32-byte boundary

    /* output
        sizeof(cacheFriendly): 64
        alignof(cacheFriendly): 32
    */
}
```

`alignof` and `_Alignof` (C11)

`_Alignof` and its alias `alignof` returns the alignment in bytes of the specified type. It returns a value of type `size_t`.

`alignof` syntax

```
alignof(type)
_Alignof(type)
```

`alignof` example

This example uses the convenience macro `alignof` because it's portable to C++. The behavior is the same if you use `_Alignof`.

```
// Compile with /std:c11

#include <stdalign.h>
#include <stdio.h>

int main()
{
    size_t alignment = alignof(short);
    printf("alignof(short) = %d\n", alignment); // 2
    printf("alignof(int) = %d\n", alignof(int)); // 4
    printf("alignof(long) = %d\n", alignof(long)); // 4
    printf("alignof(float) = %d\n", alignof(float)); // 4
    printf("alignof(double) = %d\n", alignof(double)); // 8

    typedef struct
    {
        int a;
        double b;
    } test;

    printf("alignof(test) = %d\n", alignof(test)); // 8 because that is the alignment of the largest element
in the structure

    /* output

       alignof(short) = 2
       alignof(int) = 4
       alignof(long) = 4
       alignof(float) = 4
       alignof(double) = 8
       alignof(test) = 8
    */
}
```

Requirements

Compile with `/std:c11`.

Windows SDK 10.0.20348.0 (version 2104) or later. See Windows SDK to download the latest SDK. For instructions to install and use the SDK for C11 and C17 development, see Install C11 and C17 support in Visual Studio.

See also

`/std` (Specify Language Standard Version)
C++ `alignof` and `alignas`
Compiler handling of data alignment

Declarations and Types

12/22/2021 • 2 minutes to read • Edit Online

This section describes the declaration and initialization of variables, functions, and types. The C language includes a standard set of basic data types. You can also add your own data types, called "derived types," by declaring new ones based on types already defined. The following topics are discussed:

- Overview of declarations
- Storage classes
- Type specifiers
- Type qualifiers
- Declarators and variable declarations
- Interpreting more complex declarators
- Initialization
- Storage of basic types
- Incomplete types
- Typedef declarations
- Extended storage-class attributes

See also

C Language Reference

Overview of Declarations

12/22/2021 • 4 minutes to read • Edit Online

A "declaration" specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for the object or function named by the identifier is called a "definition." C declarations for variables, functions, and types have this syntax:

Syntax

`declaration` :
  `declaration-specifiers` `attribute-seq` opt `init-declarator-list` opt `;`

/* `attribute-seq` opt is Microsoft-specific */

`declaration-specifiers` :
  `storage-class-specifier` `declaration-specifiers` opt
  `type-specifier` `declaration-specifiers` opt
  `type-qualifier` `declaration-specifiers` opt

`init-declarator-list` :
  `init-declarator`
  `init-declarator-list` `,` `init-declarator`

`init-declarator` :
  `declarator`
  `declarator` `=` `initializer`

NOTE

This syntax for `declaration` is not repeated in the following sections. Syntax in the following sections usually begins with the `declarator` nonterminal.

The declarations in the `init-declarator-list` contain the identifiers being named; `init` is an abbreviation for initializer. The `init-declarator-list` is a comma-separated sequence of declarators, each of which can have additional type information, or an initializer, or both. The `declarator` contains the identifiers, if any, being declared. The `declaration-specifiers` nonterminal consists of a sequence of type and storage-class specifiers that indicate the linkage, storage duration, and at least part of the type of the entities that the declarators denote. Declarations are made up of some combination of storage-class specifiers, type specifiers, type qualifiers, declarators, and initializers.

Declarations can contain one or more of the optional attributes listed in `attribute-seq` ; `seq` is an abbreviation for sequence. These Microsoft-specific attributes perform several functions, which are discussed in detail throughout this book.

In the general form of a variable declaration, `type-specifier` gives the data type of the variable. The `type-specifier` can be a compound, as when the type is modified by `const` or `volatile` . The `declarator` gives the name of the variable, possibly modified to declare an array or a pointer type. For example,

```
int const *fp;
```

declares a variable named `fp` as a pointer to a nonmodifiable (`const`) `int` value. You can define more than one variable in a declaration by using multiple declarators, separated by commas.

A declaration must have at least one declarator, or its type specifier must declare a structure tag, union tag, or members of an enumeration. Declarators provide any remaining information about an identifier. A declarator is an identifier that can be modified with brackets (`[ ]`), asterisks (`*`), or parentheses (`( )`) to declare an array, pointer, or function type, respectively. When you declare simple variables (such as character, integer, and floating-point items), or structures and unions of simple variables, the `declarator` is just an identifier. For more information on declarators, see Declarators and Variable Declarations.

All definitions are implicitly declarations, but not all declarations are definitions. For example, variable declarations that begin with the `extern` storage-class specifier are "referencing," rather than "defining" declarations. If an external variable is to be referred to before it's defined, or if it's defined in another source file from the one where it's used, an `extern` declaration is necessary. Storage is not allocated by "referencing" declarations, nor can variables be initialized in declarations.

A storage class or a type (or both) is required in variable declarations. Except for `__declspec`, only one storage-class specifier is allowed in a declaration and not all storage-class specifiers are permitted in every context. The `__declspec` storage class is allowed with other storage-class specifiers, and it's allowed more than once. The storage-class specifier of a declaration affects how the declared item is stored and initialized, and which parts of a program can reference the item.

The *`storage-class-specifier`* terminals defined in C include `auto`, `extern`, `register`, `static`, and `typedef`. Microsoft C also includes the *`storage-class-specifier`* terminal `__declspec`. All *`storage-class-specifier`* terminals except `typedef` and `__declspec` are discussed in Storage Classes. For information about `typedef`, see `typedef` Declarations. For information about `__declspec`, see Extended Storage-Class Attributes.

The location of the declaration within the source program and the presence or absence of other declarations of the variable are important factors in determining the lifetime of variables. There can be multiple redeclarations but only one definition. However, a definition can appear in more than one translation unit. For objects with internal linkage, this rule applies separately to each translation unit, because internally linked objects are unique to a translation unit. For objects with external linkage, this rule applies to the entire program. For more information about visibility, see Lifetime, Scope, Visibility, and Linkage.

Type specifiers provide some information about the data types of identifiers. The default type specifier is `int`. For more information, see Type Specifiers. Type specifiers can also define type tags, structure and union component names, and enumeration constants. For more information, see Enumeration Declarations, Structure Declarations, and Union Declarations.

There are two *`type-qualifier`* terminals: `const` and `volatile`. These qualifiers specify additional properties of types that are relevant only when accessing objects of that type through l-values. For more information on `const` and `volatile`, see Type Qualifiers. For a definition of l-values, see L-Value and R-Value Expressions.

See also

C Language Syntax Summary
Declarations and Types
Summary of Declarations

C Storage Classes

12/22/2021 • 2 minutes to read • Edit Online

The "storage class" of a variable determines whether the item has a "global" or "local" lifetime. C calls these two lifetimes "static" and "automatic." An item with a global lifetime exists and has a value throughout the execution of the program. All functions have global lifetimes.

Automatic variables, or variables with local lifetimes, are allocated new storage each time execution control passes to the block in which they are defined. When execution returns, the variables no longer have meaningful values.

C provides the following storage-class specifiers:

Syntax

storage-class-specifier:
`auto`
`register`
`static`
`extern`
`typedef`
`__declspec (` *extended-decl-modifier-seq* `)` /* Microsoft-specific */

Except for `__declspec`, you can use only one *storage-class-specifier* in the *declaration-specifier* in a declaration. If no storage-class specification is made, declarations within a block create automatic objects.

Items declared with the `auto` or `register` specifier have local lifetimes. Items declared with the `static` or `extern` specifier have global lifetimes.

Since `typedef` and `__declspec` are semantically different from the other four *storage-class-specifier* terminals, they are discussed separately. For specific information on `typedef`, see `typedef` Declarations. For specific information on `__declspec`, see Extended Storage-Class Attributes.

The placement of variable and function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to appear at the "external level." Declarations within function definitions appear at the "internal level."

The exact meaning of each storage-class specifier depends on two factors:

- Whether the declaration appears at the external or internal level
- Whether the item being declared is a variable or a function

Storage-Class Specifiers for External-Level Declarations and Storage-Class Specifiers for Internal-Level Declarations describe the *storage-class-specifier* terminals in each kind of declaration and explain the default behavior when the *storage-class-specifier* is omitted from a variable. Storage-Class Specifiers with Function Declarations discusses storage-class specifiers used with functions.

See also

Declarations and Types

Storage-Class Specifiers for External-Level Declarations

12/22/2021 • 4 minutes to read • Edit Online

External variables are variables at file scope. They are defined outside any function, and they are potentially available to many functions. Functions can only be defined at the external level and, therefore, cannot be nested. By default, all references to external variables and functions of the same name are references to the same object, which means they have *external linkage*. (You can use the `static` keyword to override this behavior.)

Variable declarations at the external level are either definitions of variables (*defining declarations*), or references to variables defined elsewhere (*referencing declarations*).

An external variable declaration that also initializes the variable (implicitly or explicitly) is a defining declaration of the variable. A definition at the external level can take several forms:

- A variable that you declare with the `static` storage-class specifier. You can explicitly initialize the `static` variable with a constant expression, as described in Initialization. If you omit the initializer, the variable is initialized to 0 by default. For example, these two statements are both considered definitions of the variable `k`.

  ```
  static int k = 16;
  static int k;
  ```

- A variable that you explicitly initialize at the external level. For example, `int j = 3;` is a definition of the variable `j`.

In variable declarations at the external level (that is, outside all functions), you can use the `static` or `extern` storage-class specifier or omit the storage-class specifier entirely. You cannot use the `auto` and `register` *`storage-class-specifier`* terminals at the external level.

Once a variable is defined at the external level, it is visible throughout the rest of the translation unit. The variable is not visible prior to its declaration in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described below.

The rules relating to `static` include:

- Variables declared outside all blocks without the `static` keyword always retain their values throughout the program. To restrict their access to a particular translation unit, you must use the `static` keyword. This gives them *internal linkage*. To make them global to an entire program, omit the explicit storage class or use the keyword `extern` (see the rules in the next list). This gives them *external linkage*. Internal and external linkage are also discussed in Linkage.

- You can define a variable at the external level only once within a program. You can define another variable with the same name and the `static` storage-class specifier in a different translation unit. Since each `static` definition is visible only within its own translation unit, no conflict occurs. It provides a useful way to hide identifier names that must be shared among functions of a single translation unit, but not visible to other translation units.

- The `static` storage-class specifier can apply to functions as well. If you declare a function `static`, its name is invisible outside of the file in which it's declared.

The rules for using `extern` are:

- The `extern` storage-class specifier declares a reference to a variable defined elsewhere. You can use an `extern` declaration to make a definition in another source file visible, or to make a variable visible before its definition in the same source file. Once you've declared a reference to the variable at the external level, the variable is visible throughout the remainder of the translation unit in which the declared reference occurs.

- For an `extern` reference to be valid, the variable it refers to must be defined once, and only once, at the external level. This definition (without the `extern` storage class) can be in any of the translation units that make up the program.

Example

The example below illustrates external declarations:

```
/******************************************************************
                      SOURCE FILE ONE
*******************************************************************/
#include <stdio.h>

extern int i;                // Reference to i, defined below
void next( void );           // Function prototype

int main()
{
    i++;
    printf_s( "%d\n", i );   // i equals 4
    next();
}

int i = 3;                  // Definition of i

void next( void )
{
    i++;
    printf_s( "%d\n", i );  // i equals 5
    other();
}

/******************************************************************
                      SOURCE FILE TWO
*******************************************************************/
#include <stdio.h>

extern int i;              // Reference to i in
                           // first source file
void other( void )
{
    i++;
    printf_s( "%d\n", i ); // i equals 6
}
```

The two source files in this example contain a total of three external declarations of `i`. Only one declaration is a "defining declaration." That declaration,

```
int i = 3;
```

defines the global variable `i` and initializes it with initial value 3. The "referencing" declaration of `i` at the top of the first source file using `extern` makes the global variable visible before its defining declaration in the file.

The referencing declaration of `i` in the second source file also makes the variable visible in that source file. If a defining instance for a variable is not provided in the translation unit, the compiler assumes there is an

```
extern int x;
```

referencing declaration and that a defining reference

```
int x = 0;
```

appears in another translation unit of the program.

All three functions, `main`, `next`, and `other`, perform the same task: they increase `i` and print it. The values 4, 5, and 6 are printed.

If the variable `i` hadn't been initialized, it would have been set to 0 automatically. In this case, the values 1, 2, and 3 would have been printed. See Initialization for information about variable initialization.

See also

C Storage Classes

Storage-Class Specifiers for Internal-Level Declarations

12/22/2021 • 2 minutes to read • Edit Online

You can use any of four `storage-class-specifier` terminals for variable declarations at the internal level. When you omit the `storage-class-specifier` from such a declaration, the default storage class is `auto`. Therefore, the keyword `auto` is rarely seen in a C program.

See also

C Storage Classes

`auto` Storage-Class Specifier

12/22/2021 • 2 minutes to read • Edit Online

The `auto` storage-class specifier declares an automatic variable, a variable with a local lifetime. An `auto` variable is visible only in the block in which it is declared. Declarations of `auto` variables can include initializers, as discussed in Initialization. Since variables with `auto` storage class are not initialized automatically, you should either explicitly initialize them when you declare them, or assign them initial values in statements within the block. The values of uninitialized `auto` variables are undefined. (A local variable of `auto` or `register` storage class is initialized each time it comes in scope if an initializer is given.)

An internal `static` variable (a static variable with local or block scope) can be initialized with the address of any external or `static` item, but not with the address of another `auto` item, because the address of an `auto` item is not a constant.

See also

`auto` Keyword

`register` storage-class specifier

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The Microsoft C/C++ compiler doesn't honor user requests for register variables. However, for portability all other semantics associated with the `register` keyword are honored by the compiler. For example, you can't apply the unary address-of operator (`&`) to a register object nor can the `register` keyword be used on arrays.

END Microsoft Specific

See also

Storage-class specifiers for internal-level declarations

static Storage-Class Specifier

12/22/2021 • 2 minutes to read • Edit Online

A variable declared at the internal level with the `static` storage-class specifier has a global lifetime but is visible only within the block in which it is declared. For constant strings, using `static` is useful because it alleviates the overhead of frequent initialization in often-called functions.

Remarks

If you do not explicitly initialize a `static` variable, it is initialized to 0 by default. Inside a function, `static` causes storage to be allocated and serves as a definition. Internal static variables provide private, permanent storage visible to only a single function.

See also

C Storage Classes
Storage classes (C++)

extern Storage-Class Specifier

12/22/2021 • 2 minutes to read • Edit Online

A variable declared with the `extern` storage-class specifier is a reference to a variable with the same name defined in another source file. It is used to make the external-level variable definition visible. A variable declared as `extern` has no storage allocated for itself; it is only a name.

Example

This example illustrates internal- and external-level declarations:

```
// Source1.c

int i = 1;

// Source2. c

#include <stdio.h>

// Refers to the i that is defined in Source1.c:
extern int i;

void func(void);

int main()
{
    // Prints 1:
    printf_s("%d\n", i);
    func();
    return;
}

void func(void)
{
    // Address of global i assigned to pointer variable:
    static int *external_i = &i;

    // This definition of i hides the global i in Source.c:
    int i = 16;

    // Prints 16, 1:
    printf_s("%d\n%d\n", i, *external_i);
}
```

In this example, the variable `i` is defined in Source1.c with an initial value of 1. An `extern` declaration in Source2.c is makes 'i' visible in that file.

In the `func` function, the address of the global variable `i` is used to initialize the `static` pointer variable `external_i`. This works because the global variable has `static` lifetime, meaning its address does not change during program execution. Next, a variable `i` is defined within the scope of `func` as a local variable with initial value 16. This definition does not affect the value of the external-level `i`, which is hidden by the use of its name for the local variable. The value of the global `i` is now accessible only through the pointer `external_i`.

See also

Storage-Class Specifiers for Internal-Level Declarations

Storage-Class Specifiers with Function Declarations

12/22/2021 • 2 minutes to read • Edit Online

You can use either the `static` or the `extern` storage-class specifier in function declarations. Functions always have global lifetimes.

Microsoft Specific

Function declarations at the internal level have the same meaning as function declarations at the external level. This means that a function is visible from its point of declaration throughout the rest of the translation unit even if it is declared at local scope.

END Microsoft Specific

The visibility rules for functions vary slightly from the rules for variables, as follows:

- A function declared to be `static` is visible only within the source file in which it is defined. Functions in the same source file can call the `static` function, but functions in other source files cannot access it directly by name. You can declare another `static` function with the same name in a different source file without conflict.
- Functions declared as `extern` are visible throughout all source files in the program (unless you later redeclare such a function as `static`). Any function can call an `extern` function.
- Function declarations that omit the storage-class specifier are `extern` by default.

Microsoft Specific

Microsoft allows redefinition of an `extern` identifier as `static`.

END Microsoft Specific

See also

C Storage Classes

C Type Specifiers

12/22/2021 • 2 minutes to read • Edit Online

Type specifiers in declarations define the type of a variable or function declaration.

Syntax

type-specifier: `void` `char` `short` `int` `long` `float` `double` `signed` `unsigned` *struct-or-union-specifier* *enum-specifier* *typedef-name*

The `signed char`, `signed int`, `signed short int`, and **signed long int** types, together with their `unsigned` counterparts and `enum`, are called *integral* types. The `float`, `double`, and `long double` type specifiers are referred to as *floating* or *floating-point* types. You can use any integral or floating-point type specifier in a variable or function declaration. If a *type-specifier* is not provided in a declaration, it is taken to be `int`.

The optional keywords `signed` and `unsigned` can precede or follow any of the integral types, except `enum`, and can also be used alone as type specifiers, in which case they are understood as `signed int` and `unsigned int`, respectively. When used alone, the keyword `int` is assumed to be `signed`. When used alone, the keywords `long` and `short` are understood as **long int** and `short int`.

Enumeration types are considered basic types. Type specifiers for enumeration types are discussed in Enumeration Declarations.

The keyword `void` has three uses: to specify a function return type, to specify an argument-type list for a function that takes no arguments, and to specify a pointer to an unspecified type. You can use the `void` type to declare functions that return no value or to declare a pointer to an unspecified type. See Arguments for information on `void` when it appears alone within the parentheses following a function name.

Microsoft Specific

Type checking is now ANSI-conforming, which means that type `short` and type `int` are distinct types. For example, this is a redefinition in the Microsoft C compiler that was accepted by previous versions of the compiler.

```
int   myfunc();
short myfunc();
```

This next example also generates a warning about indirection to different types:

```
int *pi;
short *ps;

ps = pi;  /* Now generates warning */
```

The Microsoft C compiler also generates warnings for differences in sign. For example:

```
signed int *pi;
unsigned int *pu

pi = pu;  /* Now generates warning */
```

Type `void` expressions are evaluated for side effects. You cannot use the (nonexistent) value of an expression that has type `void` in any way, nor can you convert a `void` expression (by implicit or explicit conversion) to any type except `void`. If you do use an expression of any other type in a context where a `void` expression is required, its value is discarded.

To conform to the ANSI specification, **void**** cannot be used as **int****. Only `void`* can be used as a pointer to an unspecified type.

END Microsoft Specific

You can create additional type specifiers with `typedef` declarations, as described in Typedef Declarations. See Storage of Basic Types for information on the size of each type.

See also

Declarations and Types

Data type specifiers and equivalents

12/22/2021 • 2 minutes to read • Edit Online

This documentation generally uses the forms of the type specifiers listed in the following table rather than the long forms. It also assumes that the `char` type is signed by default. Throughout this documentation, `char` is equivalent to `signed char`.

Type specifiers and equivalents

TYPE SPECIFIER	EQUIVALENT(S)
`signed char` [1]	`char`
`signed int`	`signed`, `int`
`signed short int`	`short`, `signed short`
`signed long int`	`long`, `signed long`
`unsigned char`	—
`unsigned int`	`unsigned`
`unsigned short int`	`unsigned short`
`unsigned long int`	`unsigned long`
`float`	—
`long double` [2]	—

[1] When you make the `char` type unsigned by default (by specifying the `/J` compiler option), you can't abbreviate `signed char` as `char`.

[2] In 32-bit and 64-bit operating systems, the Microsoft C compiler maps `long double` to type `double`.

Microsoft specific

You can specify the `/J` compiler option to change the default `char` type from `signed char` to `unsigned char`. When this option is in effect, `char` means the same as `unsigned char`, and you must use the `signed` keyword to declare a signed character value. If a `char` value is explicitly declared `signed`, the `/J` option doesn't affect it, and the value is sign-extended when widened to an `int` type. The `char` type is zero-extended when widened to `int` type.

END Microsoft specific

See also

C Type Specifiers

Type Qualifiers

12/22/2021 • 3 minutes to read • Edit Online

Type qualifiers give one of two properties to an identifier. The `const` type qualifier declares an object to be nonmodifiable. The `volatile` type qualifier declares an item whose value can legitimately be changed by something beyond the control of the program in which it appears, such as a concurrently executing thread.

The type qualifiers, `const`, `restrict`, and `volatile`, can appear only once in a declaration. Type qualifiers can appear with any type specifier; however, they can't appear after the first comma in a multiple item declaration. For example, the following declarations are legal:

```
typedef volatile int VI;
const int ci;
```

These declarations aren't legal:

```
typedef int *i, volatile *vi;
float f, const cf;
```

Type qualifiers are relevant only when accessing identifiers as l-values in expressions. See L-Value and R-Value Expressions for information about l-values and expressions.

Syntax

`type-qualifier`:
  `const`
  `restrict`
  `volatile`

`const` and `volatile`

The following are legal `const` and `volatile` declarations:

```
int const *p_ci;      // Pointer to constant int
int const (*p_ci);   // Pointer to constant int
int *const cp_i;     // Constant pointer to int
int (*const cp_i);   // Constant pointer to int
int volatile vint;     // Volatile integer
```

If the specification of an array type includes type qualifiers, the element is qualified, not the array type. If the specification of the function type includes qualifiers, the behavior is undefined. `volatile` and `const` don't affect the range of values or arithmetic properties of the object.

- The `const` keyword can be used to modify any fundamental or aggregate type, or a pointer to an object of any type, or a `typedef`. If an item is declared with only the `const` type qualifier, its type is taken to be **const int**. A `const` variable can be initialized or can be placed in a read-only region of storage. The `const` keyword is useful for declaring pointers to `const` since this requires the function not to change the pointer in any way.

- The compiler assumes that, at any point in the program, a `volatile` variable can be accessed by an unknown process that uses or modifies its value. Regardless of the optimizations specified on the command line, the code for each assignment to or reference of a `volatile` variable must be generated even if it appears to have no effect.

If `volatile` is used alone, `int` is assumed. The `volatile` type specifier can be used to provide reliable access to special memory locations. Use `volatile` with data objects that may be accessed or altered by signal handlers, by concurrently executing programs, or by special hardware such as memory-mapped I/O control registers. You can declare a variable as `volatile` for its lifetime, or you can cast a single reference to be `volatile`.

- An item can be both `const` and `volatile`, in which case the item couldn't be legitimately modified by its own program, but could be modified by some asynchronous process.

`restrict`

The `restrict` type qualifier, introduced in C99 and available in `/std:c11` or `/std:c17` mode, can be applied to pointer declarations. It qualifies the pointer, not what it points at.

`restrict` is an optimization hint to the compiler that no other pointer in the current scope refers to the same memory location. That is, only the pointer or a value derived from it (such as pointer + 1) is used to access the object during the lifetime of the pointer. This helps the compiler produce more optimized code. C++ has an equivalent mechanism, `__restrict`

Keep in mind that `restrict` is a contract between you and the compiler. If you do alias a pointer marked with `restrict`, the result is undefined.

Here's an example that uses `restrict`:

```
void test(int* restrict first, int* restrict second, int* val)
{
    *first += *val;
    *second += *val;
}

int main()
{
    int i = 1, j = 2, k = 3;
    test(&i, &j, &k);

    return 0;
}

// Marking union members restrict tells the compiler that
// only z.x or z.y will be accessed in any scope, which allows
// the compiler to optimize access to the members.
union z
{
    int* restrict x;
    double* restrict y;
};
```

See also

`/std` (Specify Language Standard Version)
Declarations and Types

Declarators and variable declarations

12/22/2021 • 2 minutes to read • Edit Online

The rest of this section describes the form and meaning of declarations for variable types summarized in this list. In particular, the remaining sections explain how to declare:

TYPE OF VARIABLE	DESCRIPTION
Simple variables	Single-value variables with integral or floating-point type
Arrays	Variables composed of a collection of elements with the same type
Pointers	Variables that point to other variables and contain variable locations (in the form of addresses) instead of values
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants
Structures	Variables composed of a collection of values that can have different types
Unions	Variables composed of several values of different types that occupy the same storage space

A *declarator* is the part of a declaration that specifies the name to introduce into the program. It can include modifiers such as `*` (pointer-to) and any of the Microsoft calling-convention keywords.

Microsoft Specific

In this declarator,

```
__declspec(thread) char *var;
```

`char` is the type specifier, `__declspec(thread)` and `*` are the modifiers, and `var` is the identifier name.

END Microsoft Specific

You use declarators to declare arrays of values, pointers to values, and functions returning values of a specified type. Declarators appear in the array and pointer declarations described later in this section.

Syntax

`declarator`:
  `pointer` opt *`direct-declarator`*

`direct-declarator`:
  `identifier`
  `(` *`declarator`* `)`
  `direct-declarator` `[` *`constant-expression`* opt `]`
  `direct-declarator` `(` *`parameter-type-list`* `)`

`direct-declarator` `(` `identifier-list` opt `)`

`pointer` :
`*` `type-qualifier-list` opt
`*` `type-qualifier-list` opt `pointer`

`type-qualifier-list` :
`type-qualifier`
`type-qualifier-list` `type-qualifier`

> **NOTE**
>
> See the syntax for `declaration` in Overview of declarations or C language syntax summary for the syntax that references a `declarator`.

When a declarator consists of an unmodified identifier, the item being declared has a base type. If an asterisk (`*`) appears to the left of an identifier, the type is modified to a pointer type. If the identifier is followed by brackets (`[ ]`), the type is modified to an array type. If parentheses follow the identifier, the type is modified to a function type. For more information about interpreting precedence within declarations, see Interpreting more complex declarators.

Each declarator declares at least one identifier. A declarator must include a type specifier to be a complete declaration. The type specifier gives: the type of the elements of an array type, the type of object addressed by a pointer type, or the return type of a function.

Array and pointer declarations are discussed in more detail later in this section. The following examples illustrate a few simple forms of declarators:

```
int list[20]; // Declares an array of 20 int values named list
char *cp;     // Declares a pointer to a char value
double func( void ); // Declares a function named func, with no
                     // arguments, that returns a double value
int *aptr[10] // Declares an array of 10 pointers
```

Microsoft Specific

The Microsoft C compiler doesn't limit the number of declarators that can modify an arithmetic, structure, or union type. The number is limited only by available memory.

END Microsoft Specific

See also

Declarations and types

Simple Variable Declarations

12/22/2021 • 2 minutes to read • Edit Online

The declaration of a simple variable, the simplest form of a direct declarator, specifies the variable's name and type. It also specifies the variable's storage class and data type.

Storage classes or types (or both) are required on variable declarations. Untyped variables (such as `var;`) generate warnings.

Syntax

declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier

identifier:
  nondigit
  identifier nondigit
  identifier digit

For arithmetic, structure, union, enumerations, and void types, and for types represented by `typedef` names, simple declarators can be used in a declaration since the type specifier supplies all the typing information. Pointer, array, and function types require more complicated declarators.

You can use a list of identifiers separated by commas (,) to specify several variables in the same declaration. All variables defined in the declaration have the same base type. For example:

```
int x, y;        /* Declares two simple variables of type int */
int const z = 1; /* Declares a constant value of type int */
```

The variables `x` and `y` can hold any value in the set defined by the `int` type for a particular implementation. The simple object `z` is initialized to the value 1 and is not modifiable.

If the declaration of `z` was for an uninitialized static variable or was at file scope, it would receive an initial value of 0, and that value would be unmodifiable.

```
unsigned long reply, flag; /* Declares two variables
                              named reply and flag     */
```

In this example, both the variables, `reply` and `flag`, have `unsigned long` type and hold unsigned integral values.

See also

Declarators and Variable Declarations

C enumeration declarations

12/22/2021 • 4 minutes to read • Edit Online

An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag. And, it defines the set of named integer identifiers (called the *enumeration set*, *enumerator constants*, *enumerators*, or *members*). A variable of the enumeration type stores one of the values of the enumeration set defined by that type.

Variables of `enum` type can be used in indexing expressions and as operands of all arithmetic and relational operators. Enumerations provide an alternative to the `#define` preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

In ANSI C, the expressions that define the value of an enumerator constant always have `int` type. That means the storage associated with an enumeration variable is the storage required for a single `int` value. An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

Syntax

`enum-specifier`:
  `enum` *`identifier`*opt `{` *`enumerator-list`* `}`
  `enum` *`identifier`*

`enumerator-list`:
  `enumerator`
  `enumerator-list` `,` *`enumerator`*

`enumerator`:
  `enumeration-constant`
  `enumeration-constant` `=` *`constant-expression`*

`enumeration-constant`:
  `identifier`

The optional *`identifier`* names the enumeration type defined by *`enumerator-list`*. This identifier is often called the "tag" of the enumeration specified by the list. A type specifier declares `identifier` to be the tag of the enumeration specified by the *`enumerator-list`* nonterminal, as seen here:

```
enum identifier
{
    // enumerator-list
}
```

The *`enumerator-list`* defines the members of the enumeration set.

If the declaration of a tag is visible, later declarations that use the tag but omit *`enumerator-list`* specify the previously declared enumerated type. The tag must refer to a defined enumeration type, and that enumeration type must be in current scope. Since the enumeration type is defined elsewhere, the *`enumerator-list`* doesn't appear in this declaration. Declarations of types derived from enumerations and `typedef` declarations for enumeration types can use the enumeration tag before the enumeration type is defined.

Each `enumeration-constant` in an `enumerator-list` names a value of the enumeration set. By default, the first `enumeration-constant` is associated with the value 0. The next `enumeration-constant` in the list is associated with the value of (`constant-expression` + 1), unless you explicitly associate it with another value. The name of an `enumeration-constant` is equivalent to its value.

You can use `enumeration-constant` = `constant-expression` to override the default sequence of values. That is, if `enumeration-constant` = `constant-expression` appears in the `enumerator-list`, the `enumeration-constant` is associated with the value given by `constant-expression`. The `constant-expression` must have `int` type and can be negative.

The following rules apply to the members of an enumeration set:

- An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers, for example, members named `null` and `zero`, in the same set.
- The identifiers in the enumeration list must be distinct from other identifiers in the same scope with the same visibility. That includes ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags obey the normal scoping rules. They must be distinct from other enumeration, structure, and union tags with the same visibility.

Examples

These examples illustrate enumeration declarations:

```
enum DAY            /* Defines an enumeration type    */
{
    saturday,       /* Names day and declares a       */
    sunday = 0,     /* variable named workday with    */
    monday,         /* that type                      */
    tuesday,
    wednesday,      /* wednesday is associated with 3 */
    thursday,
    friday
} workday;
```

The value 0 is associated with `saturday` by default. The identifier `sunday` is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In this example, a value from the set `DAY` is assigned to the variable `today`.

```
enum DAY today = wednesday;
```

The name of the enumeration constant is used to assign the value. Since the `DAY` enumeration type was previously declared, only the enumeration tag `DAY` is necessary.

To explicitly assign an integer value to a variable of an enumerated data type, use a type cast:

```
workday = ( enum DAY ) ( day_value - 1 );
```

This cast is recommended in C but isn't required.

```C
enum BOOLEAN  /* Declares an enumeration data type called BOOLEAN */
{
    false,     /* false = 0, true = 1 */
    true
};

enum BOOLEAN end_flag, match_flag; /* Two variables of type BOOLEAN */
```

This declaration can also be specified as

```C
enum BOOLEAN { false, true } end_flag, match_flag;\
```

or as

```C
enum BOOLEAN { false, true } end_flag;
enum BOOLEAN match_flag;
```

An example that uses these variables might look like this:

```C
if ( match_flag == false )
    {
     .
     .   /* statement */
     .
    }
    end_flag = true;
```

Unnamed enumerator data types can also be declared. The name of the data type is omitted, but variables can be declared. The variable `response` is a variable of the type defined:

```C
enum { yes, no } response;
```

See also

Enumerations

Structure Declarations

12/22/2021 • 4 minutes to read • Edit Online

A "structure declaration" names a type and specifies a sequence of variable values (called "members" or "fields" of the structure) that can have different types. An optional identifier, called a "tag," gives the name of the structure type and can be used in subsequent references to the structure type. A variable of that structure type holds the entire sequence defined by that type. Structures in C are similar to the types known as "records" in other languages.

Syntax

struct-or-union-specifier:
  struct-or-union identifier$_{\text{opt}}$ **{** *struct-declaration-list* **}**
  struct-or-union identifier

struct-or-union:
  `struct`
  `union`

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

struct-declaration:
  specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list:
  type-specifier specifier-qualifier-list$_{\text{opt}}$
  type-qualifier specifier-qualifier-list$_{\text{opt}}$

struct-declarator-list:
  struct-declarator struct-declarator-list **,** *struct-declarator*

struct-declarator:
  declarator
  type-specifier declarator$_{\text{opt}}$ **:** *constant-expression*

The declaration of a structure type does not set aside space for a structure. It is only a template for later declarations of structure variables.

A previously defined *identifier* (tag) can be used to refer to a structure type defined elsewhere. In this case, *struct-declaration-list* cannot be repeated as long as the definition is visible. Declarations of pointers to structures and typedefs for structure types can use the structure tag before the structure type is defined. However, the structure definition must be encountered prior to any actual use of the size of the fields. This is an incomplete definition of the type and the type tag. For this definition to be completed, a type definition must appear later in the same scope.

The *struct-declaration-list* specifies the types and names of the structure members. A *struct-declaration-list* argument contains one or more variable or bit-field declarations.

Each variable declared in *struct-declaration-list* is defined as a member of the structure type. Variable declarations within *struct-declaration-list* have the same form as other variable declarations discussed in this section, except that the declarations cannot contain storage-class specifiers or initializers. The structure members can have any variable types except type `void`, an incomplete type, or a function type.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears as long as the structure type has a tag. This allows you to create linked lists of structures.

Structures follow the same scoping as other identifiers. Structure identifiers must be distinct from other structure, union, and enumeration tags with the same visibility.

Each *struct-declaration* in a *struct-declaration-list* must be unique within the list. However, identifier names in a *struct-declaration-list* do not have to be distinct from ordinary variable names or from identifiers in other structure declaration lists.

Nested structures can also be accessed as though they were declared at the file-scope level. For example, given this declaration:

```
struct a
{
    int x;
    struct b
    {
      int y;
    } var2;
} var1;
```

these declarations are both legal:

```
struct a var3;
struct b var4;
```

Examples

These examples illustrate structure declarations:

```
struct employee   /* Defines a structure variable named temp */
{
    char name[20];
    int id;
    long class;
} temp;
```

The `employee` structure has three members: `name`, `id`, and `class`. The `name` member is a 20-element array, and `id` and `class` are simple members with `int` and `long` type, respectively. The identifier `employee` is the structure identifier.

```
struct employee student, faculty, staff;
```

This example defines three structure variables: `student`, `faculty`, and `staff`. Each structure has the same list of three members. The members are declared to have the structure type `employee`, defined in the previous example.

```
struct           /* Defines an anonymous struct and a */
{                /* structure variable named complex  */
    float x, y;
} complex;
```

The `complex` structure has two members with `float` type, `x` and `y`. The structure type has no tag and is therefore unnamed or anonymous.

```
struct sample   /* Defines a structure named x */
{
    char c;
    float *pf;
    struct sample *next;
} x;
```

The first two members of the structure are a `char` variable and a pointer to a `float` value. The third member, `next`, is declared as a pointer to the structure type being defined (`sample`).

Anonymous structures can be useful when the tag named is not needed. This is the case when one declaration defines all structure instances. For example:

```
struct
{
    int x;
    int y;
} mystruct;
```

Embedded structures are often anonymous.

```
struct somestruct
{
    struct    /* Anonymous structure */
    {
        int x, y;
    } point;
    int type;
} w;
```

Microsoft Specific

The compiler allows an unsized or zero-sized array as the last member of a structure. This can be useful if the size of a constant array differs when used in various situations. The declaration of such a structure looks like this:

`struct` *identifier* { *set-of-declarations type array-name*[]; };

Unsized arrays can appear only as the last member of a structure. Structures containing unsized array declarations can be nested within other structures as long as no further members are declared in any enclosing structures. Arrays of such structures are not allowed. The `sizeof` operator, when applied to a variable of this type or to the type itself, assumes 0 for the size of the array.

Structure declarations can also be specified without a declarator when they are members of another structure or union. The field names are promoted into the enclosing structure. For example, a nameless structure looks like this:

```
struct s
{
    float y;
    struct
    {
        int a, b, c;
    };
    char str[10];
} *p_s;
.
.
.
p_s->b = 100;  /* A reference to a field in the s structure */
```

See Structure and Union Members for information about structure references.

END Microsoft Specific

See also

Declarators and Variable Declarations

C Bit Fields

12/22/2021 • 3 minutes to read • Edit Online

In addition to declarators for members of a structure or union, a structure declarator can also be a specified number of bits, called a "bit field." Its length is set off from the declarator for the field name by a colon. A bit field is interpreted as an integral type.

Syntax

struct-declarator:
  declarator
  type-specifier declarator$_{opt}$: *constant-expression*

The *constant-expression* specifies the width of the field in bits. The *type-specifier* for the `declarator` must be `unsigned int`, `signed int`, or `int`, and the *constant-expression* must be a nonnegative integer value. If the value is zero, the declaration has no `declarator`. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional `declarator` names the bit field. Bit fields can only be declared as part of a structure. The address-of operator (&) cannot be applied to bit-field components.

Unnamed bit fields cannot be referenced, and their contents at run time are unpredictable. They can be used as "dummy" fields, for alignment purposes. An unnamed bit field whose width is specified as 0 guarantees that storage for the member following it in the *struct-declaration-list* begins on an `int` boundary.

Bit fields must also be long enough to contain the bit pattern. For example, these two statements are not legal:

```
short a:17;        /* Illegal! */
int long y:33;     /* Illegal! */
```

This example defines a two-dimensional array of structures named `screen`.

```
struct
{
    unsigned short icon : 8;
    unsigned short color : 4;
    unsigned short underline : 1;
    unsigned short blink : 1;
} screen[25][80];
```

The array contains 2,000 elements. Each element is an individual structure containing four bit-field members: `icon`, `color`, `underline`, and `blink`. The size of each structure is two bytes.

Bit fields have the same semantics as the integer type. This means a bit field is used in expressions in exactly the same way as a variable of the same base type would be used, regardless of how many bits are in the bit field.

Microsoft Specific

Bit fields defined as `int` are treated as `signed`. A Microsoft extension to the ANSI C standard allows `char` and `long` types (both `signed` and `unsigned`) for bit fields. Unnamed bit fields with base type `long`, `short`, or `char` (`signed` or `unsigned`) force alignment to a boundary appropriate to the base type.

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

```
struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;

int main( void )
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
    return 0;
}
```

the bits of `test` would be arranged as follows:

```
00000001 11110010
cccccccb bbbbaaaa
```

Since the 8086 family of processors stores the low byte of integer values before the high byte, the integer `0x01F2` above would be stored in physical memory as `0xF2` followed by `0x01`.

The ISO C99 standard lets an implementation choose whether a bit field may straddle two storage instances. Consider this structure, which stores four bit fields that total 64 bits:

```
struct
{
    unsigned int first : 9;
    unsigned int second : 7;
    unsigned int may_straddle : 30;
    unsigned int last : 18;
} tricky_bits;
```

A standard C implementation could pack these bit fields into two 32-bit integers. It might store `tricky_bits.may_straddle` as 16 bits in one 32-bit integer and 14 bits in the next 32-bit integer. The Windows ABI convention packs bit fields into single storage integers, and doesn't straddle storage units. The Microsoft compiler stores each bit field in the above example so it fits completely in a single 32-bit integer. In this case, `first` and `second` are stored in one integer, `may_straddle` is stored in a second integer, and `last` is stored in a third integer. The `sizeof` operator returns `12` on an instance of `tricky_bits`. For more information, see Padding and alignment of structure members.

END Microsoft Specific

See also

Structure Declarations

Storage and Alignment of Structures

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.

Every data object has an *alignment-requirement*. For structures, the requirement is the largest of its members. Every object is allocated an *offset* so that

offset `%` *alignment-requirement* `==` 0

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit fields.

To conserve space or to conform to existing data structures, you may want to store structures more or less compactly. The /Zp[*n*] compiler option and the #pragma pack control how structure data is "packed" into memory. When you use the /Zp[*n*] option, where *n* is 1, 2, 4, 8, or 16, each structure member after the first is stored on byte boundaries that are either the alignment requirement of the field or the packing size (*n*), whichever is smaller. Expressed as a formula, the byte boundaries are the

```
min( n, sizeof( item ) )
```

where *n* is the packing size expressed with the /Zp[*n*] option and *item* is the structure member. The default packing size is /Zp8.

To use the `pack` pragma to specify packing other than the packing specified on the command line for a particular structure, give the `pack` pragma, where the packing size is 1, 2, 4, 8, or 16, before the structure. To reinstate the packing given on the command line, specify the `pack` pragma with no arguments.

Bit fields default to size `long` for the Microsoft C compiler. Structure members are aligned on the size of the type or the /Zp[*n*] size, whichever is smaller. The default size is 4.

END Microsoft Specific

See also

Structure Declarations

Union Declarations

12/22/2021 • 2 minutes to read • Edit Online

A "union declaration" specifies a set of variable values and, optionally, a tag naming the union. The variable values are called "members" of the union and can have different types. Unions are similar to "variant records" in other languages.

Syntax

struct-or-union-specifier:
  struct-or-union identifier$_{opt}$ **{** *struct-declaration-list* **}**
  struct-or-union identifier

struct-or-union:
  `struct`
  `union`

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

The union content is defined to be

struct-declaration:
  specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list:
  type-specifier specifier-qualifier-list$_{opt}$
  type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
  struct-declarator
  struct-declarator-list **,** *struct-declarator*

A variable with `union` type stores one of the values defined by that type. The same rules govern structure and union declarations. Unions can also have bit fields.

Members of unions cannot have an incomplete type, type `void`, or function type. Therefore members cannot be an instance of the union but can be pointers to the union type being declared.

A union type declaration is a template only. Memory is not reserved until the variable is declared.

> **NOTE**
>
> If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable. For example, a union of `float` and `int` is declared. A `float` value is stored, but the program later accesses the value as an `int`. In such a situation, the value would depend on the internal storage of `float` values. The integer value would not be reliable.

Examples

The following are examples of unions:

```
union sign   /* A definition and a declaration */
{
    int svar;
    unsigned uvar;
} number;
```

This example defines a union variable with `sign` type and declares a variable named `number` that has two members: `svar`, a signed integer, and `uvar`, an unsigned integer. This declaration allows the current value of `number` to be stored as either a signed or an unsigned value. The tag associated with this union type is `sign`.

```
union               /* Defines a two-dimensional */
{                   /*  array named screen */
    struct
    {
      unsigned int icon : 8;
      unsigned color : 4;
    } window1;
    int screenval;
} screen[25][80];
```

The `screen` array contains 2,000 elements. Each element of the array is an individual union with two members: `window1` and `screenval`. The `window1` member is a structure with two bit-field members, `icon` and `color`. The `screenval` member is an `int`. At any given time, each union element holds either the `int` represented by `screenval` or the structure represented by `window1`.

Microsoft Specific

Nested unions can be declared anonymously when they are members of another structure or union. This is an example of a nameless union:

```
struct str
{
    int a, b;
    union            / * Unnamed union */
    {
      char c[4];
      long l;
      float f;
   };
   char c_array[10];
} my_str;
.
.
.
my_str.l == 0L;  /* A reference to a field in the my_str union */
```

Unions are often nested within a structure that includes a field giving the type of data contained in the union at any particular time. This is an example of a declaration for such a union:

```
struct x
{
    int type_tag;
    union
    {
      int x;
      float y;
    }
}
```

See Structure and Union Members for information about referencing unions.

END Microsoft Specific

See also

Declarators and Variable Declarations

Storage of Unions

12/22/2021 • 2 minutes to read • Edit Online

The storage associated with a union variable is the storage required for the largest member of the union. When a smaller member is stored, the union variable can contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member. For example:

```
union         /* Defines a union named x */
{
    char *a, b;
    float f[20];
} x;
```

The members of the `x` union are, in order of their declaration, a pointer to a `char` value, a `char` value, and an array of `float` values. The storage allocated for `x` is the storage required for the 20-element array `f`, since `f` is the longest member of the union. Because no tag is associated with the union, its type is unnamed or "anonymous."

See also

Union Declarations

Array Declarations

12/22/2021 • 2 minutes to read • Edit Online

An "array declaration" names the array and specifies the type of its elements. It can also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements.

Syntax

declaration:
  declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**

init-declarator-list:
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator:
  declarator
  declarator **=** *initializer*

declarator:
  pointer$_{\text{opt}}$ *direct-declarator*

direct-declarator: /* A function declarator */
  direct-declarator **[** *constant-expression*$_{\text{opt}}$ **]**

Because *constant-expression* is optional, the syntax has two forms:

- The first form defines an array variable. The *constant-expression* argument within the brackets specifies the number of elements in the array. The *constant-expression*, if present, must have integral type, and a value larger than zero. Each element has the type given by *type-specifier*, which can be any type except `void`. An array element cannot be a function type.
- The second form declares a variable that has been defined elsewhere. It omits the *constant-expression* argument in brackets, but not the brackets. You can use this form only if you previously have initialized the array, declared it as a parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

In both forms, *direct-declarator* names the variable and can modify the variable's type. The brackets (**[ ]**) following *direct-declarator* modify the declarator to an array type.

Type qualifiers can appear in the declaration of an object of array type, but the qualifiers apply to the elements rather than the array itself.

You can declare an array of arrays (a "multidimensional" array) by following the array declarator with a list of bracketed constant expressions in this form:

> *type-specifier declarator* **[** *constant-expression* **]** **[** *constant-expression* **]** ...

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. You can omit the first constant expression if you have initialized the array, declared it as a parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

You can define arrays of pointers to various types of objects by using complex declarators, as described in Interpreting More Complex Declarators.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, as described in Postfix Operators.

Examples

These examples illustrate array declarations:

```
float matrix[10][15];
```

The two-dimensional array named `matrix` has 150 elements, each having `float` type.

```
struct {
    float x, y;
} complex[100];
```

This is a declaration of an array of structures. This array has 100 elements; each element is a structure containing two members.

```
extern char *name[];
```

This statement declares the type and name of an array of pointers to `char`. The actual definition of `name` occurs elsewhere.

Microsoft Specific

The type of integer required to hold the maximum size of an array is the size of **size_t**. Defined in the header file STDDEF.H, **size_t** is an `unsigned int` with the range 0x00000000 to 0x7CFFFFFF.

END Microsoft Specific

See also

Declarators and Variable Declarations

Storage of Arrays

12/22/2021 • 2 minutes to read • Edit Online

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last.

See also

Array Declarations

Pointer Declarations

12/22/2021 • 4 minutes to read • Edit Online

A *pointer declaration* names a pointer variable and specifies the type of the object to which the variable points. A variable declared as a pointer holds a memory address.

Syntax

declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier
  (*declarator* **)**
  direct-declarator **[** *constant-expression*$_{opt}$ **]**
  direct-declarator **(** *parameter-type-list* **)**
  direct-declarator **(** *identifier-list*$_{opt}$ **)**

pointer:
  ***** *type-qualifier-list*$_{opt}$
  ***** *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
  type-qualifier
  type-qualifier-list type-qualifier

The *type-specifier* gives the type of the object, which can be any basic, structure, or union type. Pointer variables can also point to functions, arrays, and other pointers. (For information on declaring and interpreting more complex pointer types, refer to Interpreting More Complex Declarators.)

By making the *type-specifier* `void`, you can delay specification of the type to which the pointer refers. Such an item is referred to as a "pointer to `void`" and is written as `void *`. A variable declared as a pointer to *void* can be used to point to an object of any type. However, to perform most operations on the pointer or on the object to which it points, the type to which it points must be explicitly specified for each operation. (Variables of type `char` * and type `void` * are assignment-compatible without a type cast.) Such conversion can be accomplished with a type cast (see Type-Cast Conversions for more information).

The *type-qualifier* can be either `const` or `volatile`, or both. These specify, respectively, that the pointer cannot be modified by the program itself (`const`), or that the pointer can legitimately be modified by some process beyond the control of the program (`volatile`). (See Type Qualifiers for more information on `const` and `volatile`.)

The *declarator* names the variable and can include a type modifier. For example, if *declarator* represents an array, the type of the pointer is modified to be a pointer to an array.

You can declare a pointer to a structure, union, or enumeration type before you define the structure, union, or enumeration type. You declare the pointer by using the structure or union tag as shown in the examples below. Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable.

Examples

The following examples illustrate pointer declarations.

```
char *message; /* Declares a pointer variable named message */
```

The *message* pointer points to a variable with `char` type.

```
int *pointers[10];  /* Declares an array of pointers */
```

The *pointers* array has 10 elements; each element is a pointer to a variable with `int` type.

```
int (*pointer)[10]; /* Declares a pointer to an array of 10 elements */
```

The *pointer* variable points to an array with 10 elements. Each element in this array has `int` type.

```
int const *x;      /* Declares a pointer variable, x,
                      to a constant value */
```

The pointer *x* can be modified to point to a different `int` value, but the value to which it points cannot be modified.

```
const int some_object = 5 ;
int other_object = 37;
int *const y = &fixed_object;
int volatile *const z = &some_object;
int *const volatile w = &some_object;
```

The variable *y* in these declarations is declared as a constant pointer to an `int` value. The value it points to can be modified, but the pointer itself must always point to the same location: the address of *fixed_object*. Similarly, *z* is a constant pointer, but it is also declared to point to an `int` whose value cannot be modified by the program. The additional specifier `volatile` indicates that although the value of the **const int** pointed to by *z* cannot be modified by the program, it could legitimately be modified by a process running concurrently with the program. The declaration of *w* specifies that the program cannot change the value pointed to and that the program cannot modify the pointer.

```
struct list *next, *previous; /* Uses the tag for list */
```

This example declares two pointer variables, *next* and *previous*, that point to the structure type *list*. This declaration can appear before the definition of the *list* structure type (see the next example), as long as the *list* type definition has the same visibility as the declaration.

```
struct list
{
    char *token;
    int count;
    struct list *next;
} line;
```

The variable *line* has the structure type named *list*. The *list* structure type has three members: the first member is a pointer to a `char` value, the second is an `int` value, and the third is a pointer to another *list* structure.

```
struct id
{
    unsigned int id_no;
    struct name *pname;
} record;
```

The variable *record* has the structure type *id*. Note that *pname* is declared as a pointer to another structure type named *name*. This declaration can appear before the *name* type is defined.

See also

Declarators and Variable Declarations

Storage of Addresses

12/22/2021 • 2 minutes to read • Edit Online

The amount of storage required for an address and the meaning of the address depend on the implementation of the compiler. Pointers to different types are not guaranteed to have the same length. Therefore, **sizeof(char *)** is not necessarily equal to **sizeof(int *)**.

Microsoft Specific

For the Microsoft C compiler, **sizeof(char *)** is equal to **sizeof(int *)**.

END Microsoft Specific

See also

Pointer Declarations

Based Pointers (C)

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

__based (C++ Reference)

For the Microsoft 32-bit and 64-bit C compilers, a based pointer is a 32-bit or 64-bit offset from a 32-bit or 64-bit pointer base. Based addressing is useful for exercising control over sections where objects are allocated, thereby decreasing the size of the executable file and increasing execution speed. In general, the form for specifying a based pointer is

> *type* **__based(** *base* **)** *declarator*

The "based on pointer" variant of based addressing enables specification of a pointer as a base. The based pointer, then, is an offset into the memory section starting at the beginning of the pointer on which it is based. Pointers based on pointer addresses are the only form of the `__based` keyword valid in 32-bit and 64-bit compilations. In such compilations, they are 32-bit or 64-bit displacements from a 32-bit or 64-bit base.

One use for pointers based on pointers is for persistent identifiers that contain pointers. A linked list that consists of pointers based on a pointer can be saved to disk, then reloaded to another place in memory, with the pointers remaining valid.

The following example shows a pointer based on a pointer.

```
void *vpBuffer;

struct llist_t
{
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

The pointer `vpBuffer` is assigned the address of memory allocated at some later point in the program. The linked list is relocated relative to the value of `vpBuffer`.

END Microsoft Specific

See also

Declarators and Variable Declarations

C Abstract Declarators

12/22/2021 • 2 minutes to read • Edit Online

An abstract declarator is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*****) always precedes the identifier in a declarator; array (**[ ]**) and function (**( )**) modifiers follow the identifier. Knowing this, you can determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly. See Interpreting More Complex Declarators for additional information and examples of complex declarators. Generally `typedef` can be used to simplify declarators. See Typedef Declarations.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

These examples illustrate abstract declarators:

```
int *         // The type name for a pointer to type int:

int *[3]      // An array of three pointers to int

int (*) [5]   // A pointer to an array of five int

int *()       // A function with no parameter specification
              // returning a pointer to int

// A pointer to a function taking no arguments and
// returning an int

int (*) ( void )

// An array of an unspecified number of constant pointers to
// functions each with one parameter that has type unsigned int
// and an unspecified number of other parameters returning an int

int (*const []) ( unsigned int, ... )
```

> **NOTE**
>
> The abstract declarator consisting of a set of empty parentheses, **( )**, is not allowed because it is ambiguous. It is impossible to determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before the parentheses (in which case it is a function type).

See also

Declarators and Variable Declarations

Interpreting More Complex Declarators

12/22/2021 • 4 minutes to read • Edit Online

You can enclose any declarator in parentheses to specify a particular interpretation of a "complex declarator." A complex declarator is an identifier qualified by more than one array, pointer, or function modifier. You can apply various combinations of array, pointer, and function modifiers to a single identifier. Generally `typedef` may be used to simplify declarations. See Typedef Declarations.

In interpreting complex declarators, brackets and parentheses (that is, modifiers to the right of the identifier) take precedence over asterisks (that is, modifiers to the left of the identifier). Brackets and parentheses have the same precedence and associate from left to right. After the declarator has been fully interpreted, the type specifier is applied as the last step. By using parentheses you can override the default association order and force a particular interpretation. Never use parentheses, however, around an identifier name by itself. This could be misinterpreted as a parameter list.

A simple way to interpret complex declarators is to read them "from the inside out," using the following four steps:

1. Start with the identifier and look directly to the right for brackets or parentheses (if any).
2. Interpret these brackets or parentheses, then look to the left for asterisks.
3. If you encounter a right parenthesis at any stage, go back and apply rules 1 and 2 to everything within the parentheses.
4. Apply the type specifier.

```
char *( *(*var)() )[10];
 ^   ^  ^ ^ ^   ^    ^
 7   6  4 2 1   3    5
```

In this example, the steps are numbered in order and can be interpreted as follows:

1. The identifier `var` is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. `char` values.

Examples

The following examples illustrate other complex declarations and show how parentheses can affect the meaning of a declaration.

```
int *var[5]; /* Array of pointers to int values */
```

The array modifier has higher priority than the pointer modifier, so `var` is declared to be an array. The pointer modifier applies to the type of the array elements; therefore, the array elements are pointers to `int` values.

```
int (*var)[5]; /* Pointer to array of int values */
```

In this declaration for `var`, parentheses give the pointer modifier higher priority than the array modifier, and `var` is declared to be a pointer to an array of five `int` values.

```
long *var( long, long ); /* Function returning pointer to long */
```

Function modifiers also have higher priority than pointer modifiers, so this declaration for `var` declares `var` to be a function returning a pointer to a `long` value. The function is declared to take two `long` values as arguments.

```
long (*var)( long, long ); /* Pointer to function returning long */
```

This example is similar to the previous one. Parentheses give the pointer modifier higher priority than the function modifier, and `var` is declared to be a pointer to a function that returns a `long` value. Again, the function takes two `long` arguments.

```
struct both       /* Array of pointers to functions */
{                 /*   returning structures         */
    int a;
    char b;
} ( *var[5] )( struct both, struct both );
```

The elements of an array cannot be functions, but this declaration demonstrates how to declare an array of pointers to functions instead. In this example, `var` is declared to be an array of five pointers to functions that return structures with two members. The arguments to the functions are declared to be two structures with the same structure type, `both`. Note that the parentheses surrounding `*var[5]` are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below:

```
/* ILLEGAL */
struct both *var[5](struct both, struct both);
```

The following statement declares an array of pointers.

```
unsigned int *(* const *name[5][10] ) ( void );
```

The `name` array has 50 elements organized in a multidimensional array. The elements are pointers to a pointer that is a constant. This constant pointer points to a function that has no parameters and returns a pointer to an unsigned type.

This next example is a function returning a pointer to an array of three `double` values.

```
double ( *var( double (*)[3] ) )[3];
```

In this declaration, a function returns a pointer to an array, since functions returning arrays are illegal. Here `var` is declared to be a function returning a pointer to an array of three `double` values. The function `var` takes one argument. The argument, like the return value, is a pointer to an array of three `double` values. The argument type is given by a complex *abstract-declarator*. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three pointers to `double` values. For a discussion and examples of abstract declarators, see Abstract Declarators.

```
union sign         /* Array of arrays of pointers */
{                  /* to pointers to unions       */
     int x;
     unsigned y;
} **var[5][5];
```

As the above example shows, a pointer can point to another pointer, and an array can contain arrays as elements. Here `var` is an array of five elements. Each element is a five-element array of pointers to pointers to unions with two members.

```
union sign *(*var[5])[5]; /* Array of pointers to arrays
                             of pointers to unions        */
```

This example shows how the placement of parentheses changes the meaning of the declaration. In this example, `var` is a five-element array of pointers to five-element arrays of pointers to unions. For examples of how to use `typedef` to avoid complex declarations, see Typedef Declarations.

See also

Declarations and Types

Initialization

12/22/2021 • 2 minutes to read • Edit Online

An "initializer" is a value or a sequence of values to be assigned to the variable being declared. You can set a variable to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable.

The following sections describe how to initialize variables of scalar, aggregate, and string types. "Scalar types" include all the arithmetic types, plus pointers. "Aggregate types" include arrays, structures, and unions.

See also

Declarations and Types

Initializing Scalar Types

12/22/2021 • 3 minutes to read • Edit Online

When initializing scalar types, the value of the `assignment-expression` is assigned to the variable. The conversion rules for assignment apply. (See Type Conversions for information on conversion rules.)

Syntax

`declaration` :
  `declaration-specifiers` `init-declarator-list` opt `;`

`declaration-specifiers` :
  `storage-class-specifier` `declaration-specifiers` opt
  `type-specifier` `declaration-specifiers` opt
  `type-qualifier` `declaration-specifiers` opt

`init-declarator-list` :
  `init-declarator`
  `init-declarator-list` `,` `init-declarator`

`init-declarator` :
  `declarator`
  `declarator` `=` `initializer` /* For scalar initialization */

`initializer` :
  `assignment-expression`

You can initialize variables of any type, provided that you obey the following rules:

- Variables declared at the file-scope level can be initialized. If you do not explicitly initialize a variable at the external level, it is initialized to 0 by default.
- A constant expression can be used to initialize any global variable declared with the `static` `storage-class-specifier`. Variables declared to be `static` are initialized when program execution begins. If you do not explicitly initialize a global `static` variable, it is initialized to 0 by default, and every member that has pointer type is assigned a null pointer.
- Variables declared with the `auto` or `register` storage-class specifier are initialized each time execution control passes to the block in which they are declared. If you omit an initializer from the declaration of an `auto` or `register` variable, the initial value of the variable is undefined. For automatic and register values, the initializer is not restricted to being a constant; it can be any expression involving previously defined values, even function calls.
- The initial values for external variable declarations and for all `static` variables, whether external or internal, must be constant expressions. (For more information, see Constant Expressions.) Since the address of any externally declared or static variable is constant, it can be used to initialize an internally declared `static` pointer variable. However, the address of an `auto` variable cannot be used as a static initializer because it may be different for each execution of the block. You can use either constant or variable values to initialize `auto` and `register` variables.
- If the declaration of an identifier has block scope, and the identifier has external linkage, the declaration cannot have an initialization.

Examples

The following examples illustrate initializations:

```
int x = 10;
```

The integer variable `x` is initialized to the constant expression `10`.

```
register int *px = 0;
```

The pointer `px` is initialized to 0, producing a "null" pointer.

```
const int c = (3 * 1024);
```

This example uses a constant expression `(3 * 1024)` to initialize `c` to a constant value that cannot be modified because of the `const` keyword.

```
int *b = &x;
```

This statement initializes the pointer `b` with the address of another variable, `x`.

```
int *const a = &z;
```

The pointer `a` is initialized with the address of a variable named `z`. However, since it is specified to be a `const`, the variable `a` can only be initialized, never modified. It always points to the same location.

```
int GLOBAL ;

int function( void )
{
    int LOCAL ;
    static int *lp = &LOCAL;   /* Illegal initialization */
    static int *gp = &GLOBAL;  /* Legal initialization   */
    register int *rp = &LOCAL; /* Legal initialization   */
}
```

The global variable `GLOBAL` is declared at the external level, so it has global lifetime. The local variable `LOCAL` has `auto` storage class and only has an address during the execution of the function in which it is declared. Therefore, attempting to initialize the `static` pointer variable `lp` with the address of `LOCAL` is not permitted. The `static` pointer variable `gp` can be initialized to the address of `GLOBAL` because that address is always the same. Similarly, `*rp` can be initialized because `rp` is a local variable and can have a non-constant initializer. Each time the block is entered, `LOCAL` has a new address, which is then assigned to `rp`.

See also

Initialization

Initializing Aggregate Types

12/22/2021 • 5 minutes to read • Edit Online

An *aggregate* type is a structure, union, or array type. If an aggregate type contains members of aggregate types, the initialization rules apply recursively.

Syntax

initializer:
  { *initializer-list* **}** /* For aggregate initialization */
  { *initializer-list* **,** **}**

initializer-list:
  initializer
  initializer-list **,** *initializer*

The *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an initializer list. Therefore, initializer lists can be nested. This form is useful for initializing aggregate members of an aggregate type, as shown in the examples in this section. However, if the initializer for an automatic identifier is a single expression, it need not be a constant expression; it merely needs to have appropriate type for assignment to the identifier.

For each initializer list, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable.

If *initializer-list* has fewer values than an aggregate type, the remaining members or elements of the aggregate type are initialized to 0. The initial value of an automatic identifier not explicitly initialized is undefined. If *initializer-list* has more values than an aggregate type, an error results. These rules apply to each embedded initializer list, as well as to the aggregate as a whole.

A structure's initializer is either an expression of the same type, or a list of initializers for its members enclosed in curly braces (**{ }**). Unnamed bit-field members are not initialized.

When a union is initialized, *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If an array has unknown size, the number of initializers determines the size of the array, and its type becomes complete. There is no way to specify repetition of an initializer in C, or to initialize an element in the middle of an array without providing all preceding values as well. If you need this operation in your program, write the routine in assembly language.

Note that the number of initializers can set the size of the array:

```
int x[ ] = { 0, 1, 2 }
```

If you specify the size and give the wrong number of initializers, however, the compiler generates an error.

Microsoft Specific

The maximum size for an array is defined by **size_t**. Defined in the header file STDDEF.H, **size_t** is an `unsigned int` with the range 0x00000000 to 0x7CFFFFFF.

END Microsoft Specific

Examples

This example shows initializers for an array.

```
int P[4][3] =
{
    { 1, 1, 1 },
    { 2, 2, 2 },
    { 3, 3, 3,},
    { 4, 4, 4,},
};
```

This statement declares `P` as a four-by-three array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row. Note that the initializer list for the third and fourth rows contains commas after the last constant expression. The last initializer list (`{4, 4, 4,},`) is also followed by a comma. These extra commas are permitted but are not required; only commas that separate constant expressions from one another, and those that separate one initializer list from another, are required.

If an aggregate member has no embedded initializer list, values are simply assigned, in order, to each member of the subaggregate. Therefore, the initialization in the previous example is equivalent to the following:

```
int P[4][3] =
{
   1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list and would help to clarify the example above.

When you initialize an aggregate variable, you must be careful to use braces and initializer lists properly. The following example illustrates the compiler's interpretation of braces in more detail:

```
typedef struct
{
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] =
{
    { {  1, 2, 3 }, {  4, 5, 6 }, {  7, 8, 9 } },  /* Row 1 */
    { { 10,11,12 }, { 13,14,15 }, { 16,17,18 } }   /* Row 2 */
};
```

In this example, `nlist` is declared as a 2-by-3 array of structures, each structure having three members. Row 1 of the initialization assigns values to the first row of `nlist`, as follows:

1. The first left brace on row 1 signals the compiler that initialization of the first aggregate member of `nlist` (that is, `nlist[0]`) is beginning.

2. The second left brace indicates that initialization of the first aggregate member of `nlist[0]` (that is, the structure at `nlist[0][0]`) is beginning.

3. The first right brace ends initialization of the structure `nlist[0][0]`; the next left brace starts initialization of `nlist[0][1]`.

4. The process continues until the end of the line, where the closing right brace ends initialization of `nlist[0]`.

Row 2 assigns values to the second row of `nlist` in a similar way. Note that the outer sets of braces enclosing the initializers on rows 1 and 2 are required. The following construction, which omits the outer braces, would cause an error:

```
triplet nlist[2][3] =  /* THIS CAUSES AN ERROR */
{
     {  1, 2, 3 },{  4, 5, 6 },{  7, 8, 9 },   /* Line 1 */
     { 10,11,12 },{ 13,14,15 },{ 16,17,18 }    /* Line 2 */
};
```

In this construction, the first left brace on line 1 starts the initialization of `nlist[0]`, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of `nlist[0]` is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, `{ 4,5,6 }` initializes the first structure in the second row of `nlist`. The remaining two structures of `nlist[1]` are set to 0. When the compiler encounters the next initializer list (`{ 7,8,9 }`), it tries to initialize `nlist[2]`. Since `nlist` has only two rows, this attempt causes an error.

In this next example, the three `int` members of `x` are initialized to 1, 2, and 3, respectively.

```
struct list
{
    int i, j, k;
    float m[2][3];
} x = {
        1,
        2,
        3,
       {4.0, 4.0, 4.0}
      };
```

In the `list` structure above, the three elements in the first row of `m` are initialized to 4.0; the elements of the remaining row of `m` are initialized to 0.0 by default.

```
union
{
    char x[2][3];
    int i, j, k;
} y = { {
            {'1'},
            {'4'}
        }
      };
```

The union variable `y`, in this example, is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list `{'1'}` assigns values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character `1`, and the remaining two elements in the row are initialized to the value 0 by default. Similarly, the first element of the second row of `x` is initialized to the character `4`, and the remaining two elements in the row are initialized to the value 0.

See also

Initialization

Initializing Strings

12/22/2021 • 2 minutes to read • Edit Online

You can initialize an array of characters (or wide characters) with a string literal (or wide string literal). For example:

```
char code[ ] = "abc";
```

initializes `code` as a four-element array of characters. The fourth element is the null character, which terminates all string literals.

An identifier list can only be as long as the number of identifiers to be initialized. If you specify an array size that is shorter than the string, the extra characters are ignored. For example, the following declaration initializes `code` as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to `code`. The character `d` and the string-terminating null character are discarded. Note that this creates an unterminated string (that is, one without a 0 value to mark its end) and generates a diagnostic message indicating this condition.

The declaration

```
char s[] = "abc", t[3] = "abc";
```

is identical to

```
char s[]  = {'a', 'b', 'c', '\0'},
     t[3] = {'a', 'b', 'c' };
```

If the string is shorter than the specified array size, the remaining elements of the array are initialized to 0.

Microsoft Specific

In Microsoft C, string literals can be up to 2048 bytes in length.

END Microsoft Specific

See also

Initialization

Storage of basic types

12/22/2021 • 2 minutes to read • Edit Online

The following table summarizes the storage associated with each basic type.

Sizes of fundamental types

TYPE	STORAGE
`char`, `unsigned char`, `signed char`	1 byte
`short`, `unsigned short`	2 bytes
`int`, `unsigned int`	4 bytes
`long`, `unsigned long`	4 bytes
`long long`, `unsigned long long`	8 bytes
`float`	4 bytes
`double`	8 bytes
`long double`	8 bytes

The C data types fall into general categories. The *integral types* include `int`, `char`, `short`, `long`, and `long long`. These types can be qualified with `signed` or `unsigned`, and `unsigned` by itself can be used as shorthand for `unsigned int`. Enumeration types (`enum`) are also treated as integral types for most purposes. The *floating types* include `float`, `double`, and `long double`. The *arithmetic types* include all floating and integral types.

See also

Declarations and types

Type char

12/22/2021 • 2 minutes to read • Edit Online

The `char` type is used to store the integer value of a member of the representable character set. That integer value is the ASCII code corresponding to the specified character.

Microsoft Specific

Character values of type `unsigned char` have a range from 0 to 0xFF hexadecimal. A `signed char` has range 0x80 to 0x7F. These ranges translate to 0 to 255 decimal, and -128 to +127 decimal, respectively. The /J compiler option changes the default from `signed` to `unsigned`.

END Microsoft Specific

See also

Storage of Basic Types

Type int

12/22/2021 • 2 minutes to read • Edit Online

The size of a `signed int` or `unsigned int` item is the standard size of an integer on a particular machine. For example, in 16-bit operating systems, the `int` type is usually 16 bits, or 2 bytes. In 32-bit operating systems, the `int` type is usually 32 bits, or 4 bytes. Thus, the `int` type is equivalent to either the `short int` or the `long int` type, and the `unsigned int` type is equivalent to either the `unsigned short` or the `unsigned long` type, depending on the target environment. The `int` types all represent signed values unless specified otherwise.

The type specifiers `int` and `unsigned int` (or simply `unsigned`) define certain features of the C language (for instance, the `enum` type). In these cases, the definitions of `int` and `unsigned int` for a particular implementation determine the actual storage.

Microsoft Specific

Signed integers are represented in two's-complement form. The most-significant bit holds the sign: 1 for negative, 0 for positive and zero. The range of values is given in C and C++ Integer Limits, which is taken from the LIMITS.H header file.

END Microsoft Specific

> **NOTE**
> The `int` and `unsigned int` type specifiers are widely used in C programs because they allow a particular machine to handle integer values in the most efficient way for that machine. However, since the sizes of the `int` and `unsigned int` types vary, programs that depend on a specific `int` size may not be portable to other machines. To make programs more portable, you can use expressions with the `sizeof` operator (as discussed in The `sizeof` Operator) instead of hard-coded data sizes.

See also

Storage of Basic Types

C Sized Integer Types

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

Microsoft C features support for sized integer types. You can declare 8-, 16-, 32-, or 64-bit integer variables by using the `__intN` type specifier, where `N` is the size, in bits, of the integer variable. The value of *n* can be 8, 16, 32, or 64. The following example declares one variable of each of the four types of sized integers:

```
__int8  nSmall;     // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

The first three types of sized integers are synonyms for the ANSI types that have the same size. They're useful for writing portable code that behaves identically across multiple platforms. The `__int8` data type is synonymous with type `char`, `__int16` is synonymous with type `short`, `__int32` is synonymous with type `int`, and `__int64` is synonymous with type `long long`.

END Microsoft Specific

See also

Storage of Basic Types

Type float

12/22/2021 • 3 minutes to read • Edit Online

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers) format. Single-precision values with float type have 4 bytes, consisting of a sign bit, an 8-bit excess-127 binary exponent, and a 23-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E-38 to 3.4E+38 for type float.

You can declare variables as float or double, depending on the needs of your application. The principal differences between the two types are the significance they can represent, the storage they require, and their range. The following table shows the relationship between significance and storage requirements.

Floating-Point Types

TYPE	SIGNIFICANT DIGITS	NUMBER OF BYTES
float	6 - 7	4
double	15 - 16	8

Floating-point variables are represented by a mantissa, which contains the value of the number, and an exponent, which contains the order of magnitude of the number.

The following table shows the number of bits allocated to the mantissa and the exponent for each floating-point type. The most significant bit of any float or double is always the sign bit. If it is 1, the number is considered negative; otherwise, it is considered a positive number.

Lengths of Exponents and Mantissas

TYPE	EXPONENT LENGTH	MANTISSA LENGTH
float	8 bits	23 bits
double	11 bits	52 bits

Because exponents are stored in an unsigned form, the exponent is biased by half its possible value. For type float, the bias is 127; for type double, it is 1023. You can compute the actual exponent value by subtracting the bias value from the exponent value.

The mantissa is stored as a binary fraction greater than or equal to 1 and less than 2. For types float and double, there is an implied leading 1 in the mantissa in the most-significant bit position, so the mantissas are actually 24 and 53 bits long, respectively, even though the most-significant bit is never stored in memory.

Instead of the storage method just described, the floating-point package can store binary floating-point numbers as denormalized numbers. "Denormalized numbers" are nonzero floating-point numbers with reserved exponent values in which the most-significant bit of the mantissa is 0. By using the denormalized format, the range of a floating-point number can be extended at the cost of precision. You cannot control whether a floating-point number is represented in normalized or denormalized form; the floating-point package determines the representation. The floating-point package never uses a denormalized form unless the exponent becomes less than the minimum that can be represented in a normalized form.

The following table shows the minimum and maximum values you can store in variables of each floating-point type. The values listed in this table apply only to normalized floating-point numbers; denormalized floating-point numbers have a smaller minimum value. Note that numbers retained in 80*x*87 registers are always represented in 80-bit normalized form; numbers can only be represented in denormalized form when stored in 32-bit or 64-bit floating-point variables (variables of type float and type long).

Range of Floating-Point Types

TYPE	MINIMUM VALUE	MAXIMUM VALUE
float	1.175494351 E - 38	3.402823466 E + 38
double	2.2250738585072014 E - 308	1.7976931348623158 E + 308

If precision is less of a concern than storage, consider using type float for floating-point variables. Conversely, if precision is the most important criterion, use type double.

Floating-point variables can be promoted to a type of greater significance (from type float to type double). Promotion often occurs when you perform arithmetic on floating-point variables. This arithmetic is always done in as high a degree of precision as the variable with the highest degree of precision. For example, consider the following type declarations:

```
float f_short;
double f_long;
long double f_longer;

f_short = f_short * f_long;
```

In the preceding example, the variable `f_short` is promoted to type double and multiplied by `f_long`; then the result is rounded to type float before being assigned to `f_short`.

In the following example (which uses the declarations from the preceding example), the arithmetic is done in float (32-bit) precision on the variables; the result is then promoted to type double:

```
f_longer = f_short * f_short;
```

See also

Storage of Basic Types

Type double

12/22/2021 • 2 minutes to read • Edit Online

Double precision values with double type have 8 bytes. The format is similar to the float format except that it has an 11-bit excess-1023 exponent and a 52-bit mantissa, plus the implied high-order 1 bit. This format gives a range of approximately 1.7E-308 to 1.7E+308 for type double.

Microsoft Specific

The double type contains 64 bits: 1 for sign, 11 for the exponent, and 52 for the mantissa. Its range is +/- 1.7E308 with at least 15 digits of precision.

END Microsoft Specific

See also

Storage of Basic Types

Type long double

12/22/2021 • 2 minutes to read • Edit Online

The `long double` type is identical to the double type.

See also

Storage of Basic Types

Incomplete Types

12/22/2021 • 2 minutes to read • Edit Online

An *incomplete type* is a type that describes an identifier but lacks information needed to determine the size of the identifier. An incomplete type can be:

- A structure type whose members you have not yet specified.
- A union type whose members you have not yet specified.
- An array type whose dimension you have not yet specified.

The `void` type is an incomplete type that cannot be completed. To complete an incomplete type, specify the missing information. The following examples show how to create and complete the incomplete types.

- To create an incomplete structure type, declare a structure type without specifying its members. In this example, the `ps` pointer points to an incomplete structure type called `student`.

```
struct student *ps;
```

- To complete an incomplete structure type, declare the same structure type later in the same scope with its members specified, as in

```
struct student
{
    int num;
}                   /* student structure now completed */
```

- To create an incomplete array type, declare an array type without specifying its repetition count. For example:

```
char a[];  /* a has incomplete type */
```

- To complete an incomplete array type, declare the same name later in the same scope with its repetition count specified, as in

```
char a[25]; /* a now has complete type */
```

See also

Declarations and Types

Typedef Declarations

12/22/2021 • 3 minutes to read • Edit Online

A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. You can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that you have declared. Typedef names allow you to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

Syntax

declaration:
  declaration-specifiers init-declarator-list$_{opt}$;

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

storage-class-specifier:
  `typedef`

type-specifier:
  `void`
  `char`
  `short`
  `int`
  `long`
  `float`
  `double`
  `signed`
  `unsigned`
  struct-or-union-specifier
  enum-specifier
  typedef-name

typedef-name:
  identifier

Note that a typedef declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. When a typedef name is used as a type specifier, it can be combined with certain type specifiers, but not others. Acceptable modifiers include `const` and `volatile`.

Typedef names share the name space with ordinary identifiers (see Name Spaces for more information). Therefore, a program can have a typedef name and a local-scope identifier by the same name. For example:

```
typedef char FlagType;

int main()
{
}

int myproc( int )
{
    int FlagType;
}
```

When declaring a local-scope identifier by the same name as a typedef, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified. This example illustrates this constraint:

```
typedef char FlagType;
const FlagType x;
```

To reuse the `FlagType` name for an identifier, a structure member, or a union member, the type must be provided:

```
const int FlagType;  /* Type specifier required */
```

It is not sufficient to say

```
const FlagType;      /* Incomplete specification */
```

because the `FlagType` is taken to be part of the type, not an identifier that is being redeclared. This declaration is taken to be an illegal declaration like

```
int;  /* Illegal declaration */
```

You can declare any type with typedef, including pointer, function, and array types. You can declare a typedef name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

Typedef names can be used to improve code readability. All three of the following declarations of `signal` specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv( int ), (*pfv)( int );  /* typedef declarations */

void ( *signal( int, void (*) (int)) ) ( int );
fv *signal( int, fv * );   /* Uses typedef type */
pfv signal( int, pfv );    /* Uses typedef type */
```

Examples

The following examples illustrate typedef declarations:

```
typedef int WHOLE; /* Declares WHOLE to be a synonym for int */
```

Note that `WHOLE` could now be used in a variable declaration such as `WHOLE i;` or `const WHOLE i;`. However, the declaration `long WHOLE i;` would be illegal.

```
typedef struct club
{
    char name[30];
    int size, year;
} GROUP;
```

This statement declares `GROUP` as a structure type with three members. Since a structure tag, `club`, is also specified, either the typedef name (`GROUP`) or the structure tag can be used in declarations. You must use the struct keyword with the tag, and you cannot use the struct keyword with the typedef name.

```
typedef GROUP *PG; /* Uses the previous typedef name
                      to declare a pointer            */
```

The type `PG` is declared as a pointer to the `GROUP` type, which in turn is defined as a structure type.

```
typedef void DRAWF( int, int );
```

This example provides the type `DRAWF` for a function returning no value and taking two int arguments. This means, for example, that the declaration

```
DRAWF box;
```

is equivalent to the declaration

```
void box( int, int );
```

See also

Declarations and Types

C extended storage-class attributes

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

More up-to-date information on storage class attributes can be found under `__declspec` (C++ Reference).

Extended attribute syntax simplifies and standardizes the Microsoft-specific extensions to the C language. The storage-class attributes that use extended attribute syntax include `thread`, `naked`, `dllimport`, and `dllexport`.

The extended attribute syntax for specifying storage-class information uses the `__declspec` keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage-class attribute (`thread` , `naked`, `dllimport`, or `dllexport`). Examples of other storage-class modifiers include the `static` and `extern` keywords. However, these keywords are part of the ISO C standard and aren't covered by extended attribute syntax.

Syntax

`storage-class-specifier`:
    `__declspec (` `extended-decl-modifier-seq` `)` /* Microsoft-specific */

`extended-decl-modifier-seq`: /* Microsoft-specific */
    `extended-decl-modifier` opt
    `extended-decl-modifier-seq` `extended-decl-modifier`

`extended-decl-modifier`: /* Microsoft-specific */
    `thread`
    `naked`
    `dllimport`
    `dllexport`

White space separates the declaration modifiers. An `extended-decl-modifier-seq` can be empty; in this case, `__declspec` has no effect.

The `thread`, `naked`, `dllimport`, and `dllexport` storage-class attributes are a property only of the declaration of the data or function to which they're applied. They don't redefine the type attributes of the function itself. The `thread` attribute affects data only. The `naked` attribute affects functions only. The `dllimport` and `dllexport` attributes affect functions and data.

END Microsoft Specific

See also

Declarations and types

DLL Import and Export

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The dllimport and dllexport storage-class modifiers are Microsoft-specific extensions to the C language. These modifiers define the DLL's interface to its client (the executable file or another DLL). For specific information about using these modifiers, see dllexport, dllimport.

END Microsoft Specific

See also

C Extended Storage-Class Attributes

Naked (C)

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The naked storage-class attribute is a Microsoft-specific extension to the C language. The compiler generates code without prolog and epilog code for functions declared with the naked storage-class attribute. Naked functions are useful when you need to write your own prolog/epilog code sequences using inline assembler code. Naked functions are useful for writing virtual device drivers.

For specific information about using the naked attribute, see Naked Functions.

END Microsoft Specific

See also

C Extended Storage-Class Attributes

Thread Local Storage

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

Thread Local Storage (TLS) is the mechanism by which each thread in a given multithreaded process allocates storage for thread-specific data. In standard multithreaded programs, data is shared among all threads of a given process, whereas thread local storage is the mechanism for allocating per-thread data. For a complete discussion of threads, see Processes and Threads in the Windows SDK.

The Microsoft C language includes the extended storage-class attribute, thread, which is used with the __declspec keyword to declare a thread local variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

These guidelines must be observed when you are declaring statically bound thread local variables:

- Thread-local variables that have dynamic initialization are only initialized on the thread that causes the DLL to load, and threads that are already running in the process. For more information, see thread.
- You can apply the thread attribute only to data declarations and definitions. It cannot be used on function declarations or definitions. For example, the following code generates a compiler error:

  ```
  #define Thread   __declspec( thread )
  Thread void func();      /* Error */
  ```

- You can specify the thread attribute only on data items with static storage duration. This includes global data (both static and extern) and local static data. You cannot declare automatic data with the thread attribute. For example, the following code generates compiler errors:

  ```
  #define Thread   __declspec( thread )
  void func1()
  {
      Thread int tls_i;            /* Error */
  }

  int func2( Thread int tls_i )    /* Error */
  {
     return tls_i;
  }
  ```

- You must use the thread attribute for the declaration and the definition of thread local data, regardless of whether the declaration and definition occur in the same file or separate files. For example, the following code generates an error:

  ```
  #define Thread   __declspec( thread )
  extern int tls_i;     /* This generates an error, because the   */
  int Thread tls_i;     /* declaration and the definition differ. */
  ```

- You cannot use the thread attribute as a type modifier. For example, the following code generates a compiler error:

```
char *ch __declspec( thread );      /* Error */
```

- The address of a thread local variable is not considered constant, and any expression involving such an address is not considered a constant expression. This means that you cannot use the address of a thread local variable as an initializer for a pointer. For example, the compiler flags the following code as an error:

```
#define Thread   __declspec( thread )
Thread int tls_i;
int *p = &tls_i;      /* Error */
```

- C permits initialization of a variable with an expression involving a reference to itself, but only for objects of nonstatic extent. For example:

```
#define Thread   __declspec( thread )
Thread int tls_i = tls_i;             /* Error */
int j = j;                            /* Error */
Thread int tls_i = sizeof( tls_i )    /* Okay  */
```

  Note that a sizeof expression that includes the variable being initialized does not constitute a reference to itself and is allowed.

- The use of **__declspec(thread)** may interfere with delay loading of DLL imports.

For more information about using the thread attribute, see Multithreading Topics.

END Microsoft Specific

See also

C Extended Storage-Class Attributes

Expressions and Assignments

12/22/2021 • 2 minutes to read • Edit Online

This section describes how to form expressions and to assign values in the C language. Constants, identifiers, strings, and function calls are all operands that are manipulated in expressions. The C language has all the usual language operators. This section covers those operators as well as operators that are unique to C or Microsoft C. The topics discussed include:

- L-value and r-value expressions
- Constant expressions
- Side effects
- Sequence points
- Operators
- Operator precedence
- Type conversions
- Type casts

See also

C Language Reference

Operands and Expressions

12/22/2021 • 2 minutes to read • Edit Online

An "operand" is an entity on which an operator acts. An "expression" is a sequence of operators and operands that performs any combination of these actions:

- Computes a value
- Designates an object or function
- Generates side effects

Operands in C include constants, identifiers, strings, function calls, subscript expressions, member-selection expressions, and complex expressions formed by combining operands with operators or by enclosing operands in parentheses. The syntax for these operands is given in Primary Expressions.

See also

Expressions and Assignments

C primary expressions

12/22/2021 • 2 minutes to read • Edit Online

Primary expressions are the building blocks of more complex expressions. They may be constants, identifiers, a Generic selection, or an expression in parentheses.

Syntax

`primary-expression` :
  `identifier`
  `constant`
  `string-literal`
  (`expression`)
  `generic-selection`

expression:
  `assignment-expression`
  `expression` , `assignment-expression`

See also

Generic selection Operands and Expressions

Identifiers in Primary Expressions

12/22/2021 • 2 minutes to read • Edit Online

Identifiers can have integral, `float`, `enum`, `struct`, `union`, array, pointer, or function type. An identifier is a primary expression provided it has been declared as designating an object (in which case it is an l-value) or as a function (in which case it is a function designator). See L-Value and R-Value Expressions for a definition of l-value.

The pointer value represented by an array identifier is not a variable, so an array identifier cannot form the left-hand operand of an assignment operation and therefore is not a modifiable l-value.

An identifier declared as a function represents a pointer whose value is the address of the function. The pointer addresses a function returning a value of a specified type. Thus, function identifiers also cannot be l-values in assignment operations. For more information, see Identifiers.

See also

C Primary Expressions

Constants in Primary Expressions

12/22/2021 • 2 minutes to read • Edit Online

A constant operand has the value and type of the constant value it represents. A character constant has `int` type. An integer constant has `int`, `long`, `unsigned int`, or `unsigned long` type, depending on the integer's size and on the way the value is specified. See Constants for more information.

See also

C Primary Expressions

String Literals in Primary Expressions

12/22/2021 • 2 minutes to read • Edit Online

A "string literal" is a character, wide character, or sequence of adjacent characters enclosed in double quotation marks. Since they are not variables, neither string literals nor any of their elements can be the left-hand operand in an assignment operation. The type of a string literal is an array of `char` (or an array of `wchar_t` for wide-string literals). Arrays in expressions are converted to pointers. See String Literals for more information about strings.

See also

C Primary Expressions

Expressions in Parentheses

12/22/2021 • 2 minutes to read • Edit Online

You can enclose any operand in parentheses without changing the type or value of the enclosed expression. For example, in the expression:

```
( 10 + 5 ) / 5
```

the parentheses around `10 + 5` mean that the value of `10 + 5` is evaluated first and it becomes the left operand of the division (/) operator. The result of `( 10 + 5 ) / 5` is 3. Without the parentheses, `10 + 5 / 5` would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation in all cases. For example, neither the parentheses nor the left-to-right grouping of the following expression guarantees what the value of `i` will be in either of the subexpressions:

```
( i++ +1 ) * ( 2 + i )
```

The compiler is free to evaluate the two sides of the multiplication in any order. If the initial value of `i` is zero, the whole expression could be evaluated as either of these two statements:

```
( 0 + 1 + 1 ) * ( 2 + 1 )
( 0 + 1 + 1 ) * ( 2 + 0 )
```

Exceptions resulting from side effects are discussed in Side Effects.

See also

C Primary Expressions

Generic selection (C11)

12/22/2021 • 2 minutes to read • Edit Online

Use the `_Generic` keyword to write code that selects an expression at compile time based on the type of the argument. It's similar to overloading in C++ where the type of the argument selects which function to call. In this case, the type of the argument selects which expression to evaluate.

For example, the expression `_Generic(42, int: "integer", char: "character", default: "unknown");` evaluates the type of `42` and looks for the matching type, `int`, in the list. It finds it and returns `"integer"`.

Syntax

`generic-selection`:
  `_Generic` (*`assignment-expression`* , *`assoc-list`*)

`assoc-list`:
  `association`
  `assoc-list` , *`association`*

`association`:
  `type-name` : *`assignment-expression`*
  `default` : *`assignment-expression`*

The first *`assignment-expression`* is called the controlling expression. The type of the controlling expression is determined at compile time and matched against the *`assoc-list`* to find which expression to evaluate and return. The controlling expression isn't evaluated. For example, `_Generic(intFunc(), int: "integer", default: "error");` doesn't result in a call at runtime to `intFunc`.

When the type of the controlling expression is determined, `const`, `volatile`, and `restrict` are removed before matching against *`assoc-list`*.

Entries in the `assoc-list` that aren't chosen aren't evaluated.

Constraints

- The *`assoc-list`* can't specify the same type more than once.
- The *`assoc-list`* can't specify types that are compatible with each other, such as an enumeration and the underlying type of that enumeration.
- If a generic selection doesn't have a default, the controlling expression must have only one compatible type name in the generic association list.

Example

One way to use `_Generic` is in a macro. The <tgmath.h> header file uses `_Generic` to call the right math function depending on the type of argument. For example, the macro for `cos` maps a call with a float to `cosf`, while mapping a call with a complex double to `ccos`.

The following example shows how to write a macro that identifies the type of the argument you pass to it. It produces `"unknown"` if no entry in the *`assoc-list`* matches the controlling expression:

```c
// Compile with /std:c11

#include <stdio.h>

/* Get a type name string for the argument x */
#define TYPE_NAME(X) _Generic((X), \
      int: "int", \
      char: "char", \
      double: "double", \
      default: "unknown")

int main()
{
    printf("Type name: %s\n", TYPE_NAME(42.42));

    // The following would result in a compile error because
    // 42.4 is a double, doesn't match anything in the list,
    // and there is no default.
    // _Generic(42.4, int: "integer", char: "character"));
}

/* Output:
Type name: double
*/
```

Requirements

Compile with `/std:c11`.

Windows SDK 10.0.20348.0 (version 2104) or later. See Windows SDK to download the latest SDK. For instructions to install and use the SDK for C11 and C17 development, see Install C11 and C17 support in Visual Studio.

See also

`/std` (Specify language standard version)
Type-generic math

L-Value and R-Value Expressions

12/22/2021 • 2 minutes to read • Edit Online

Expressions that refer to memory locations are called "l-value" expressions. An l-value represents a storage region's "locator" value, or a "left" value, implying that it can appear on the left of the equal sign (=). L-values are often identifiers.

Expressions referring to modifiable locations are called "modifiable l-values." A modifiable l-value cannot have an array type, an incomplete type, or a type with the `const` attribute. For structures and unions to be modifiable l-values, they must not have any members with the `const` attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location.

An identifier is a modifiable l-value if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if `ptr` is a pointer to a storage region, then `*ptr` is a modifiable l-value that designates the storage region to which `ptr` points.

Any of the following C expressions can be l-value expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript ([ ]) expression that does not evaluate to an array
- A member-selection expression (-> or .)
- A unary-indirection (*) expression that does not refer to an array
- An l-value expression in parentheses
- A `const` object (a nonmodifiable l-value)

The term "r-value" is sometimes used to describe the value of an expression and to distinguish it from an l-value. All l-values are r-values but not all r-values are l-values.

Microsoft Specific

Microsoft C includes an extension to the ANSI C standard that allows casts of l-values to be used as l-values, as long as the size of the object is not lengthened through the cast. (See Type-Cast Conversions for more information.) The following example illustrates this feature:

```
char *p ;
short  i;
long l;

(long *) p = &l ;       /* Legal cast   */
(long) i = l ;          /* Illegal cast */
```

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za compiler option to disable these extensions.

END Microsoft Specific

See also

Operands and Expressions

C Constant Expressions

12/22/2021 • 2 minutes to read • Edit Online

A constant expression gets evaluated at compile time, not run time, and can be used in any place that a constant can be used. The constant expression must evaluate to a constant that is in the range of representable values for that type. The operands of a constant expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts, `sizeof` expressions, and other constant expressions.

Syntax

`constant-expression` :
  `conditional-expression`

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

`expression` :
  `assignment-expression`
  `expression` `,` *`assignment-expression`*

`assignment-expression` :
  `conditional-expression`
  `unary-expression` *`assignment-operator`* *`assignment-expression`*

`assignment-operator` : one of
  `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

The nonterminals for struct declarator, enumerator, direct declarator, direct-abstract declarator, and labeled statement contain the *`constant-expression`* nonterminal.

An integral constant expression must be used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a `case` constant.

Constant expressions used in preprocessor directives are subject to several restrictions. They're known as *restricted* constant expressions. A restricted constant expression can't contain `sizeof` expressions, enumeration constants, type casts to any type, or floating-type constants. It can, however, contain the special constant expression `defined (` *identifier* `)`.

See also

Operands and Expressions

Expression Evaluation (C)

12/22/2021 • 2 minutes to read • Edit Online

Expressions involving assignment, unary increment, unary decrement, or calling a function may have consequences incidental to their evaluation (side effects). When a "sequence point" is reached, everything preceding the sequence point, including any side effects, is guaranteed to have been evaluated before evaluation begins on anything following the sequence point.

"Side effects" are changes caused by the evaluation of an expression. Side effects occur whenever the value of a variable is changed by an expression evaluation. All assignment operations have side effects. Function calls can also have side effects if they change the value of an externally visible item, either by direct assignment or by indirect assignment through a pointer.

See also

Operands and Expressions

Side Effects

12/22/2021 • 2 minutes to read • Edit Online

The order of evaluation of expressions is defined by the specific implementation, except when the language guarantees a particular order of evaluation (as outlined in Precedence and Order of Evaluation). For example, side effects occur in the following function calls:

```
add( i + 1, i = j + 2 );
myproc( getc(), getc() );
```

The arguments of a function call can be evaluated in any order. The expression `i + 1` may be evaluated before `i = j + 2`, or `i = j + 2` may be evaluated before `i + 1`. The result is different in each case. Likewise, it is not possible to guarantee what characters are actually passed to the `myproc`. Since unary increment and decrement operations involve assignments, such operations can cause side effects, as shown in the following example:

```
x[i] = i++;
```

In this example, the value of `x` that is modified is unpredictable. The value of the subscript could be either the new or the old value of `i`. The result can vary under different compilers or different optimization levels.

Since C does not define the order of evaluation of side effects, both evaluation methods discussed above are correct and either may be implemented. To make sure that your code is portable and clear, avoid statements that depend on a particular order of evaluation for side effects.

See also

Expression Evaluation

C Sequence Points

12/22/2021 • 2 minutes to read • Edit Online

Between consecutive "sequence points" an object's value can be modified only once by an expression. The C language defines the following sequence points:

- Left operand of the logical-AND operator (&&). The left operand of the logical-AND operator is completely evaluated and all side effects complete before continuing. If the left operand evaluates to false (0), the other operand is not evaluated.
- Left operand of the logical-OR operator (`||`). The left operand of the logical-OR operator is completely evaluated and all side effects complete before continuing. If the left operand evaluates to true (nonzero), the other operand is not evaluated.
- Left operand of the comma operator. The left operand of the comma operator is completely evaluated and all side effects complete before continuing. Both operands of the comma operator are always evaluated. Note that the comma operator in a function call does not guarantee an order of evaluation.
- Function-call operator. All arguments to a function are evaluated and all side effects complete before entry to the function. No order of evaluation among the arguments is specified.
- First operand of the conditional operator. The first operand of the conditional operator is completely evaluated and all side effects complete before continuing.
- The end of a full initialization expression (that is, an expression that is not part of another expression such as the end of an initialization in a declaration statement).
- The expression in an expression statement. Expression statements consist of an optional expression followed by a semicolon (;). The expression is evaluated for its side effects and there is a sequence point following this evaluation.
- The controlling expression in a selection (`if` or `switch`) statement. The expression is completely evaluated and all side effects complete before the code dependent on the selection is executed.
- The controlling expression of a `while` or `do` statement. The expression is completely evaluated and all side effects complete before any statements in the next iteration of the `while` or `do` loop are executed.
- Each of the three expressions of a `for` statement. The expressions are completely evaluated and all side effects complete before any statements in the next iteration of the `for` loop are executed.
- The expression in a `return` statement. The expression is completely evaluated and all side effects complete before control returns to the calling function.

See also

Expression Evaluation

C Operators

12/22/2021 • 2 minutes to read • Edit Online

The C operators are a subset of the C++ built-in operators.

There are three types of operators. A unary expression consists of either a unary operator prepended to an operand, or the `sizeof` keyword followed by an expression. The expression can be either the name of a variable or a cast expression. If the expression is a cast expression, it must be enclosed in parentheses. A binary expression consists of two operands joined by a binary operator. A ternary expression consists of three operands joined by the conditional-expression operator.

C includes the following unary operators:

SYMBOL	NAME
- ~ !	Negation and complement operators
`*` `&`	Indirection and address-of operators
`sizeof`	Size operator
+	Unary plus operator
++ --	Unary increment and decrement operators

Binary operators associate from left to right. C provides the following binary operators:

SYMBOL	NAME
`*` `/` `%`	Multiplicative operators
+ -	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

The base operator (:>), supported by previous versions of the Microsoft 16-bit C compiler, is described in C Language Syntax Summary.

The conditional-expression operator has lower precedence than binary expressions and differs from them in being right associative.

Expressions with operators also include assignment expressions, which use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators; the binary assignment operators are the simple-assignment operator (=) and the compound-assignment operators. Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator.

See also

- Expressions and Assignments

Precedence and order of evaluation

12/22/2021 • 4 minutes to read • Edit Online

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. Precedence can also be described by the word "binding." Operators with a higher precedence are said to have tighter binding.

The following table summarizes the precedence and associativity (the order in which the operands are evaluated) of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together, they have equal precedence and are evaluated according to their associativity. The operators in the table are described in the sections beginning with Postfix Operators. The rest of this section gives general information about precedence and associativity.

Precedence and associativity of C operators

SYMBOL [1]	TYPE OF OPERATION	ASSOCIATIVITY
`[` `]` `(` `)` `.` `->` `++` `--` (postfix)	Expression	Left to right
`sizeof` `&` `*` `+` `-` `~` `!` `++` `--` (prefix)	Unary	Right to left
typecasts	Unary	Right to left
`*` `/` `%`	Multiplicative	Left to right
`+` `-`	Additive	Left to right
`<<` `>>`	Bitwise shift	Left to right
`<` `>` `<=` `>=`	Relational	Left to right
`==` `!=`	Equality	Left to right
`&`	Bitwise-AND	Left to right
`^`	Bitwise-exclusive-OR	Left to right
`|`	Bitwise-inclusive-OR	Left to right
`&&`	Logical-AND	Left to right
`||`	Logical-OR	Left to right
`? :`	Conditional-expression	Right to left

SYMBOL	TYPE OF OPERATION	ASSOCIATIVITY
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	Simple and compound assignment [2]	Right to left
`,`	Sequential evaluation	Left to right

[1] Operators are listed in descending order of precedence. If several operators appear on the same line or in a group, they have equal precedence.

[2] All simple and compound-assignment operators have equal precedence.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right. The direction of evaluation does not affect the results of expressions that include more than one multiplication (`*`), addition (`+`), or binary-bitwise (`&`, `|`, or `^`) operator at the same level. Order of operations is not defined by the language. The compiler is free to evaluate such expressions in any order, if the compiler can guarantee a consistent result.

Only the sequential-evaluation (`,`), logical-AND (`&&`), logical-OR (`||`), conditional-expression (`? :`), and function-call operators constitute sequence points, and therefore guarantee a particular order of evaluation for their operands. The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (`,`) is guaranteed to evaluate its operands from left to right. (The comma operator in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.) For more information, see Sequence points.

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. This is called "short-circuit" evaluation. Thus, some operands of the expression may not be evaluated. For example, in the expression

`x && y++`

the second operand, `y++`, is evaluated only if `x` is true (nonzero). Thus, `y` is not incremented if `x` is false (0).

Examples

The following list shows how the compiler automatically binds several sample expressions:

EXPRESSION	AUTOMATIC BINDING
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

In the first expression, the bitwise-AND operator (`&`) has higher precedence than the logical-OR operator (`||`), so `a & b` forms the first operand of the logical-OR operation.

In the second expression, the logical-OR operator (`||`) has higher precedence than the simple-assignment operator (`=`), so `b || c` is grouped as the right-hand operand in the assignment. Note that the value assigned to `a` is either 0 or 1.

The third expression shows a correctly formed expression that may produce an unexpected result. The logical-

AND operator (`&&`) has higher precedence than the logical-OR operator (`||`), so `q && r` is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, `q && r` is evaluated before `s--` . However, if `q && r` evaluates to a nonzero value, `s--` is not evaluated, and `s` is not decremented. If not decrementing `s` would cause a problem in your program, `s--` should appear as the first operand of the expression, or `s` should be decremented in a separate operation.

The following expression is illegal and produces a diagnostic message at compile time:

ILLEGAL EXPRESSION	DEFAULT GROUPING
`p == 0 ? p += 1: p += 2`	`( p == 0 ? p += 1 : p ) += 2`

In this expression, the equality operator (`==`) has the highest precedence, so `p == 0` is grouped as an operand. The conditional-expression operator (`? :`) has the next-highest precedence. Its first operand is `p == 0`, and its second operand is `p += 1` . However, the last operand of the conditional-expression operator is considered to be `p` rather than `p += 2`, since this occurrence of `p` binds more closely to the conditional-expression operator than it does to the compound-assignment operator. A syntax error occurs because `+= 2` does not have a left-hand operand. You should use parentheses to prevent errors of this kind and produce more readable code. For example, you could use parentheses as shown below to correct and clarify the preceding example:

```
( p == 0 ) ? ( p += 1 ) : ( p += 2 )
```

See also

C operators

Usual Arithmetic Conversions

12/22/2021 • 2 minutes to read • Edit Online

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating types. These conversions are known as "arithmetic conversions." Conversion of an operand value to a compatible type causes no change to its value.

The arithmetic conversions summarized below are called "usual arithmetic conversions." These steps are applied only for binary operators that expect arithmetic type. The purpose is to yield a common type which is also the type of the result. To determine which conversions actually take place, the compiler applies the following algorithm to binary operations in the expression. The steps below are not a precedence order.

1. If either operand is of type `long double`, the other operand is converted to type `long double`.
2. If the above condition is not met and either operand is of type `double`, the other operand is converted to type `double`.
3. If the above two conditions are not met and either operand is of type `float`, the other operand is converted to type `float`.
4. If the above three conditions are not met (none of the operands are of floating types), then integral conversions are performed on the operands as follows:
   - If either operand is of type `unsigned long`, the other operand is converted to type `unsigned long`.
   - If the above condition is not met and either operand is of type `long` and the other of type `unsigned int`, both operands are converted to type `unsigned long`.
   - If the above two conditions are not met, and either operand is of type `long`, the other operand is converted to type `long`.
   - If the above three conditions are not met, and either operand is of type `unsigned int`, the other operand is converted to type `unsigned int`.
   - If none of the above conditions are met, both operands are converted to type `int`.

The following code illustrates these conversion rules:

```
float   fVal;
double  dVal;
int   iVal;
unsigned long ulVal;

dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Uses step 4.
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Uses step 3.
                      * Result of addition converted to double
                      */
```

See also

C Operators

Postfix Operators

12/22/2021 • 2 minutes to read • Edit Online

The postfix operators have the highest precedence (the tightest binding) in expression evaluation.

Syntax

postfix-expression:
  primary-expression
  postfix-expression **[** *expression* **]**
  postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
  postfix-expression **.** *identifier*
  postfix-expression **->** *identifier*
  postfix-expression **++**
  postfix-expression **--**

Operators in this precedence level are the array subscripts, function calls, structure and union members, and postfix increment and decrement operators.

See also

C Operators

One-Dimensional Arrays

12/22/2021 • 2 minutes to read • Edit Online

A postfix expression followed by an expression in square brackets ([ ]) is a subscripted representation of an element of an array object. A subscript expression represents the value at the address that is *expression* positions beyond *postfix-expression* when expressed as

```
postfix-expression [ expression ]
```

Usually, the value represented by *postfix-expression* is a pointer value, such as an array identifier, and *expression* is an integral value. However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *postfix-expression* position and the pointer value could be in the brackets in the *expression*, or "subscript," position. For example, this code is legal:

```
// one_dimensional_arrays.c
int sum, *ptr, a[10];
int main() {
   ptr = a;
   sum = 4[ptr];
}
```

Subscript expressions are generally used to refer to array elements, but you can apply a subscript to any pointer. Whatever the order of values, *expression* must be enclosed in brackets ([ ]).

The subscript expression is evaluated by adding the integral value to the pointer value, then applying the indirection operator (*) to the result. (See Indirection and Address-of Operators for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that `a` is a pointer and `b` is an integer:

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules for the addition operator (given in Additive Operators), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier `line` refers to an array of `int` values. The following procedure is used to evaluate the subscript expression `line[ i ]`:

1. The integer value `i` is multiplied by the number of bytes defined as the length of an `int` item. The converted value of `i` represents `i` `int` positions.

2. This converted value is added to the original pointer value (`line`) to yield an address that is offset `i` `int` positions from `line`.

3. The indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, `line [ i ]`).

The subscript expression `line[0]` represents the value of the first element of line, since the offset from the address represented by `line` is 0. Similarly, an expression such as `line[5]` refers to the element offset five positions from line, or the sixth element of the array.

See also

Subscript Operator:

Multidimensional Arrays (C)

12/22/2021 • 2 minutes to read • Edit Online

A subscript expression can also have multiple subscripts, as follows:

```
expression1 [ expression2 ] [ expression3 ] ...
```

Subscript expressions associate from left to right. The leftmost subscript expression, *expression1* **[** *expression2* **]**, is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*****) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type (see examples below).

Expressions with multiple subscripts refer to elements of "multidimensional arrays." A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions.

Examples

For the following examples, an array named `prop` is declared with three elements, each of which is a 4-by-6 array of `int` values.

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
```

A reference to the `prop` array looks like this:

```
i = prop[0][0][1];
```

The example above shows how to refer to the second individual `int` element of `prop`. Arrays are stored by row, so the last subscript varies most quickly; the expression `prop[0][0][2]` refers to the next (third) element of the array, and so on.

```
i = prop[2][1][3];
```

This statement is a more complex reference to an individual element of `prop`. The expression is evaluated as follows:

1. The first subscript, `2`, is multiplied by the size of a 4-by-6 `int` array and added to the pointer value `prop`. The result points to the third 4-by-6 array of `prop`.
2. The second subscript, `1`, is multiplied by the size of the 6-element `int` array and added to the address represented by `prop[2]`.
3. Each element of the 6-element array is an `int` value, so the final subscript, `3`, is multiplied by the size of an `int` before it is added to `prop[2][1]`. The resulting pointer addresses the fourth element of the 6-element array.
4. The indirection operator is applied to the pointer value. The result is the `int` element at that address.

These next two examples show cases where the indirection operator is not applied.

```
ip = prop[2][1];

ipp = prop[2];
```

In the first of these statements, the expression `prop[2][1]` is a valid reference to the three-dimensional array `prop`; it refers to a 6-element array (declared above). Since the pointer value addresses an array, the indirection operator is not applied.

Similarly, the result of the expression `prop[2]` in the second statement `ipp = prop[2];` is a pointer value addressing a two-dimensional array.

See also

Subscript Operator:

Function Call (C)

12/22/2021 • 2 minutes to read • Edit Online

A *function call* is an expression that includes the name of the function being called or the value of a function pointer and, optionally, the arguments being passed to the function.

Syntax

postfix-expression:
  postfix-expression **(** *argument-expression-list*$_{\text{opt}}$ **)**

argument-expression-list:
  assignment-expression
  argument-expression-list **,** *assignment-expression*

The *postfix-expression* must evaluate to a function address (for example, a function identifier or the value of a function pointer), and *argument-expression-list* is a list of expressions (separated by commas) whose values (the "arguments") are passed to the function. The *argument-expression-list* argument can be empty.

A function-call expression has the value and type of the function's return value. A function cannot return an object of array type. If the function's return type is `void` (that is, the function has been declared never to return a value), the function-call expression also has `void` type. (See Function Calls for more information.)

See also

Function Call Operator: ()

Structure and Union Members

12/22/2021 • 2 minutes to read • Edit Online

A "member-selection expression" refers to members of structures and unions. Such an expression has the value and type of the selected member.

> *postfix-expression* . *identifier*
> *postfix-expression* -> *identifier*

This list describes the two forms of the member-selection expressions:

1. In the first form, *postfix-expression* represents a value of `struct` or `union` type, and *identifier* names a member of the specified structure or union. The value of the operation is that of *identifier* and is an l-value if *postfix-expression* is an l-value. See L-Value and R-Value Expressions for more information.

2. In the second form, *postfix-expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union. The value is that of *identifier* and is an l-value.

The two forms of member-selection expressions have similar effects.

In fact, an expression involving the member-selection operator (**->**) is a shorthand version of an expression using the period (**.**) if the expression before the period consists of the indirection operator (*****) applied to a pointer value. Therefore,

```
expression->identifier
```

is equivalent to

```
(*expression).identifier
```

when *expression* is a pointer value.

Examples

The following examples refer to this structure declaration. For information about the indirection operator (*****) used in these examples, see Indirection and Address-of Operators.

```
struct pair
{
    int a;
    int b;
    struct pair *sp;
} item, list[10];
```

A member-selection expression for the `item` structure looks like this:

```
item.sp = &item;
```

In the example above, the address of the `item` structure is assigned to the `sp` member of the structure. This means that `item` contains a pointer to itself.

```
(item.sp)->a = 24;
```

In this example, the pointer expression `item.sp` is used with the member-selection operator (->) to assign a value to the member `a`.

```
list[8].b = 12;
```

This statement shows how to select an individual structure member from an array of structures.

See also

Member Access Operators: . and ->

C Postfix Increment and Decrement Operators

12/22/2021 • 2 minutes to read • Edit Online

Operands of the postfix increment and decrement operators are scalar types that are modifiable l-values.

Syntax

postfix-expression:
  postfix-expression ++
  postfix-expression --

The result of the postfix increment or decrement operation is the value of the operand. After the result is obtained, the value of the operand is incremented (or decremented). The following code illustrates the postfix increment operator.

```
if( var++ > 0 )
    *p++ = *q++;
```

In this example, the variable `var` is compared to 0, then incremented. If `var` was positive before being incremented, the next statement is executed. First, the value of the object pointed to by `q` is assigned to the object pointed to by `p`. Then, `q` and `p` are incremented.

See also

Postfix Increment and Decrement Operators: ++ and --

C Unary Operators

12/22/2021 • 2 minutes to read • Edit Online

Unary operators appear before their operand and associate from right to left.

Syntax

unary-expression: *postfix-expression*

++ *unary-expression*

`--` *unary-expression*

unary-operator cast-expression

`sizeof` *unary-expression*

sizeof (*type-name* **)**

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

See also

C Operators

Prefix Increment and Decrement Operators

12/22/2021 • 2 minutes to read • Edit Online

The unary operators (`++` and --) are called "prefix" increment or decrement operators when the increment or decrement operators appear before the operand. Postfix increment and decrement has higher precedence than prefix increment and decrement. The operand must have integral, floating, or pointer type and must be a modifiable l-value expression (an expression without the `const` attribute). The result is an l-value.

When the operator appears before its operand, the operand is incremented or decremented and its new value is the result of the expression.

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

Example

This example illustrates the unary prefix decrement operator:

```
if( line[--i] != '\n' )
    return;
```

In this example, the variable `i` is decremented before it is used as a subscript to `line`.

See also

C Unary Operators

Indirection and Address-of Operators

12/22/2021 • 2 minutes to read • Edit Online

The unary indirection operator (`*`) accesses a value indirectly, through a pointer. The operand must be a pointer type. The result of the operation is the value addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses.

The result of the indirection operator is *type* if the operand is of type *pointer to type*. If the operand points to a function, the result is a function designator. If it points to an object, the result is an lvalue that designates the object.

If the pointer value is not valid, the result of the indirection operator is undefined. These are some of the most common conditions that invalidate a pointer value:

- The pointer is a null pointer.
- The pointer specifies the address of an object after the end of its lifetime (such as an object that's gone out of scope or that's been deallocated) at the time of the reference.
- The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
- The pointer specifies an address not used by the executing program.

The unary address-of operator (`&`) gives the address of its operand. The operand must be either an lvalue that designates an object that is not declared `register` and is not a bit-field, or the result of a unary `*` operator or an array dereference (`[]`) operator, or a function designator. The result is of type *pointer to type* for an operand of type *type*.

If the operand is the result of a unary `*` operator, neither operator is evaluated and the result is as if both were omitted. The result is not an lvalue, and the constraints on the operators still apply. If the operand is the result of a `[]` operator, neither the `&` operator nor the unary `*` implied by the `[]` operator is evaluated. The result has the same effect as removing the `&` operator and changing the `[]` operator to a `+` operator. Otherwise, the result is a pointer to the object or function designated by the operand.

Examples

The following examples use these common declarations:

```
int *pa, x;
int a[20];
double d;
```

This statement uses the address-of operator (`&`) to take the address of the sixth element of the array `a` . The result is stored in the pointer variable `pa` :

```
pa = &a[5];
```

The indirection operator (`*`) is used in this example to access the `int` value at the address stored in `pa` . The value is assigned to the integer variable `x` :

```
x = *pa;
```

This example demonstrates that the result of applying the indirection operator to the address of `x` is the same as `x`:

```
assert( x == *&x );
```

This example shows equivalent ways of declaring a pointer to a function:

```
int roundup( void );     /* Function declaration */

int  *proundup  = roundup;
int  *pround  = &roundup;
assert( pround == proundup );
```

Once the function `roundup` is declared, two pointers to `roundup` are declared and initialized. The first pointer, `proundup`, is initialized using only the name of the function, while the second, `pround`, uses the address-of operator in the initialization. The initializations are equivalent.

See also

Indirection Operator: `*`
Address-of Operator: &

Unary Arithmetic Operators

12/22/2021 • 2 minutes to read • Edit Online

The C unary plus, arithmetic-negation, complement, and logical-negation operators are discussed in the following list:

OPERATOR	DESCRIPTION
+	The unary plus operator preceding an expression in parentheses forces the grouping of the enclosed operations. It is used with expressions involving more than one associative or commutative binary operator. The operand must have arithmetic type. The result is the value of the operand. An integral operand undergoes integral promotion. The type of the result is the type of the promoted operand.
-	The arithmetic-negation operator produces the negative (two's complement) of its operand. The operand must be an integral or floating value. This operator performs the usual arithmetic conversions.
`~`	The bitwise-complement (or bitwise-NOT) operator produces the bitwise complement of its operand. The operand must be of integral type. This operator performs usual arithmetic conversions; the result has the type of the operand after conversion.
!	The logical-negation (logical-NOT) operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has `int` type. The operand must be an integral, floating, or pointer value.

Unary arithmetic operations on pointers are illegal.

Examples

The following examples illustrate the unary arithmetic operators:

```
short x = 987;
    x = -x;
```

In the example above, the new value of `x` is the negative of 987, or -987.

```
unsigned short y = 0xAAAA;
    y = ~y;
```

In this example, the new value assigned to `y` is the one's complement of the unsigned value 0xAAAA, or 0x5555.

```
if( !(x < y) )
```

If `x` is greater than or equal to `y`, the result of the expression is 1 (true). If `x` is less than `y`, the result is 0 (false).

See also

Expressions with Unary Operators

sizeof Operator (C)

12/22/2021 • 2 minutes to read • Edit Online

The `sizeof` operator gives the amount of storage, in bytes, required to store an object of the type of the operand. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

Syntax

```
sizeof unary-expression
sizeof ( type-name )
```

Remarks

The operand is either an identifier that is a *unary-expression*, or a type-cast expression (that is, a type specifier enclosed in parentheses). The *unary-expression* cannot represent a bit-field object, an incomplete type, or a function designator. The result is an unsigned integral constant. The standard header STDDEF.H defines this type as **size_t**.

When you apply the `sizeof` operator to an array identifier, the result is the size of the entire array rather than the size of the pointer represented by the array identifier.

When you apply the `sizeof` operator to a structure or union type name, or to an identifier of structure or union type, the result is the number of bytes in the structure or union, including internal and trailing padding. This size may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the individual members.

If an unsized array is the last element of a structure, the `sizeof` operator returns the size of the structure without the array.

```
buffer = calloc(100, sizeof (int) );
```

This example uses the `sizeof` operator to pass the size of an `int`, which varies among machines, as an argument to a run-time function named `calloc`. The value returned by the function is stored in `buffer`.

```
static char *strings[] = {
      "this is string one",
      "this is string two",
      "this is string three",
   };
const int string_no = ( sizeof strings ) / ( sizeof strings[0] );
```

In this example, `strings` is an array of pointers to `char`. The number of pointers is the number of elements in the array, but is not specified. It is easy to determine the number of pointers by using the `sizeof` operator to calculate the number of elements in the array. The `const` integer value `string_no` is initialized to this number. Because it is a `const` value, `string_no` cannot be modified.

See also

C Operators

C++ Built-in Operators, Precedence and Associativity

Cast Operators

12/22/2021 • 2 minutes to read • Edit Online

A type cast provides a method for explicit conversion of the type of an object in a specific situation.

Syntax

cast-expression: *unary-expression*

(*type-name* **)** *cast-expression*

The compiler treats *cast-expression* as type *type-name* after a type cast has been made. Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions, discussed in Assignment Conversions. Additional restraints on casts may result from the actual sizes or representation of specific types. See Storage of Basic Types for information on actual sizes of integral types. For more information on type casts, see Type-Cast Conversions.

See also

Cast Operator: ()

C Multiplicative Operators

12/22/2021 • 2 minutes to read • Edit Online

The multiplicative operators perform multiplication (*****), division (**/**), and remainder (**%**) operations.

Syntax

multiplicative-expression: *cast-expression* *multiplicative-expression* ***** *cast-expression* *multiplicative-expression* **/** *cast-expression* *multiplicative-expression* **%** *cast-expression*

The operands of the remainder operator (**%**) must be integral. The multiplication (*****) and division (**/**) operators can take integral- or floating-type operands; the types of the operands can be different.

The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

> **NOTE**
> Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

The C multiplicative operators are described below:

OPERATOR	DESCRIPTION
*****	The multiplication operator causes its two operands to be multiplied.
/	The division operator causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules: - The result of division by 0 is undefined according to the ANSI C standard. The Microsoft C compiler generates an error at compile time or run time. - If both operands are positive or unsigned, the result is truncated toward 0. - If either operand is negative, whether the result of the operation is the largest integer less than or equal to the algebraic quotient or is the smallest integer greater than or equal to the algebraic quotient is implementation defined. (See the Microsoft-specific section below.)

OPERATOR	DESCRIPTION
%	The result of the remainder operator is the remainder when the first operand is divided by the second. When the division is inexact, the result is determined by the following rules: - If the right operand is zero, the result is undefined. - If both operands are positive or unsigned, the result is positive. - If either operand is negative and the result is inexact, the result is implementation defined. (See the Microsoft-specific section below.)

Microsoft-specific

In division where either operand is negative, the direction of truncation is toward 0.

If either operation is negative in division with the remainder operator, the result has the same sign as the dividend (the first operand in the expression).

Examples

The declarations shown below are used for the following examples:

```
int i = 10, j = 3, n;
double x = 2.0, y;
```

This statement uses the multiplication operator:

```
y = x * i;
```

In this case, `x` is multiplied by `i` to give the value 20.0. The result has `double` type.

```
n = i / j;
```

In this example, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

```
n = i % j;
```

This statement assigns `n` the integer remainder, 1, when 10 is divided by 3.

Microsoft Specific

The sign of the remainder is the same as the sign of the dividend. For example:

```
50 % -6 = 2
-50 % 6 = -2
```

In each case, `50` and `2` have the same sign.

END Microsoft Specific

See also

Multiplicative Operators and the Modulus Operator

C Additive Operators

12/22/2021 • 2 minutes to read • Edit Online

The additive operators perform addition (**+**) and subtraction (**-**).

Syntax

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

> **NOTE**
>
> Although the syntax for *additive-expression* includes *multiplicative-expression*, this does not imply that expressions using multiplication are required. See the syntax in C Language Syntax Summary, for *multiplicative-expression*, *cast-expression*, and *unary-expression*.

The operands can be integral or floating values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator.

The additive operators perform the usual arithmetic conversions on integral and floating operands. The type of the result is the type of the operands after conversion. Since the conversions performed by the additive operators do not provide for overflow or underflow conditions, information may be lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

See also

Additive Operators: + and -

Addition (+)

12/22/2021 • 2 minutes to read • Edit Online

The addition operator (**+**) causes its two operands to be added. Both operands can be either integral or floating types, or one operand can be a pointer and the other an integer.

When an integer is added to a pointer, the integer value (*i*) is converted by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value representing the address *i* positions from the original address. The new pointer value addresses a value of the same type as the original pointer value and therefore is the same as array indexing (see One-Dimensional Arrays and Multidimensional Arrays). If the sum pointer points outside the array, except at the first location beyond the high end, the result is undefined. For more information, see Pointer Arithmetic.

See also

C Additive Operators

Subtraction (-)

12/22/2021 • 2 minutes to read • Edit Online

The subtraction operator (-) subtracts the second operand from the first. Both operands can be either integral or floating types, or one operand can be a pointer and the other an integer.

When two pointers are subtracted, the difference is converted to a signed integral value by dividing the difference by the size of a value of the type that the pointers address. The size of the integral value is defined by the type `ptrdiff_t` in the standard include file STDDEF.H. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed in Pointer Arithmetic.

When an integer value is subtracted from a pointer value, the subtraction operator converts the integer value (*i*) by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is subtracted from the pointer value, the result is the memory address *i* positions before the original address. The new pointer points to a value of the type addressed by the original pointer value.

See also

C Additive Operators

Using the Additive Operators

12/22/2021 • 2 minutes to read • Edit Online

The following examples, which illustrate the addition and subtraction operators, use these declarations:

```
int i = 4, j;
float x[10];
float *px;
```

These statements are equivalent:

```
px = &x[4 + i];
px = &x[4] + i;
```

The value of `i` is multiplied by the length of a `float` and added to `&x[4]`. The resulting pointer value is the address of `x[8]`.

```
j = &x[i] - &x[i-2];
```

In this example, the address of the third element of `x` (given by `x[i-2]`) is subtracted from the address of the fifth element of `x` (given by `x[i]`). The difference is divided by the length of a `float`; the result is the integer value 2.

See also

C Additive Operators

Pointer Arithmetic

12/22/2021 • 2 minutes to read • Edit Online

Additive operations involving a pointer and an integer give meaningful results only if the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. When the integer value is converted to an address offset, the compiler assumes that only memory positions of the same size lie between the original address and the address plus the offset.

This assumption is valid for array members. By definition, an array is a series of values of the same type; its elements reside in contiguous memory locations. However, storage for any types except array elements is not guaranteed to be filled by the same type of identifiers. That is, blanks can appear between memory positions, even positions of the same type. Therefore, the results of adding to or subtracting from the addresses of any values but array elements are undefined.

Similarly, when two pointer values are subtracted, the conversion assumes that only values of the same type, with no blanks, lie between the addresses given by the operands.

See also

C Additive Operators

Bitwise Shift Operators

12/22/2021 • 2 minutes to read • Edit Online

The shift operators shift their first operand left (`<<`) or right (`>>`) by the number of positions the second operand specifies.

Syntax

shift-expression:
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is `unsigned`, they are set to 0. Otherwise, they are filled with copies of the sign bit. For left-shift operators without overflow, the statement

```
expr1 << expr2
```

is equivalent to multiplication by 2^{expr2}. For right-shift operators,

```
expr1 >> expr2
```

is equivalent to division by 2^{expr2} if `expr1` is unsigned or has a nonnegative value.

The result of a shift operation is undefined if the second operand is negative, or if the right operand is greater than or equal to the width in bits of the promoted left operand.

Since the conversions performed by the shift operators do not provide for overflow or underflow conditions, information may be lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

```
unsigned int x, y, z;

x = 0x00AA;
y = 0x5500;

z = ( x << 8 ) + ( y >> 8 );
```

In this example, `x` is shifted left eight positions and `y` is shifted right eight positions. The shifted values are added, giving 0xAA55, and assigned to `z`.

Shifting a negative value to the right yields half the original value, rounded down. For example, -253 (binary 11111111 00000011) shifted right one bit produces -127 (binary 11111111 10000001). A positive 253 shifts right to produce +126.

Right shifts preserve the sign bit. When a signed integer shifts right, the most-significant bit remains set. When an unsigned integer shifts right, the most-significant bit is cleared.

See also

Left Shift and Right Shift Operators (>> and <<)

C Relational and Equality Operators

12/22/2021 • 3 minutes to read • Edit Online

The binary relational and equality operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false. The type of the result is `int`.

Syntax

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

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

The relational and equality operators test the following relationships:

OPERATOR	RELATIONSHIP TESTED
`<`	First operand less than second operand
`>`	First operand greater than second operand
`<=`	First operand less than or equal to second operand
`>=`	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The first four operators in the list above have a higher precedence than the equality operators (`==` and `!=`). See the precedence information in the table Precedence and Associativity of C Operators.

The operands can have integral, floating, or pointer type. The types of the operands can be different. Relational operators perform the usual arithmetic conversions on integral and floating type operands. In addition, you can use the following combinations of operand types with the relational and equality operators:

- Both operands of any relational or equality operator can be pointers to the same type. For the equality (`==`) and inequality (`!=`) operators, the result of the comparison indicates whether the two pointers address the same memory location. For the other relational operators (<, >, <=, and >=), the result of the comparison indicates the relative position of the two memory addresses of the objects pointed to. Relational operators compare only offsets.

  Pointer comparison is defined only for parts of the same object. If the pointers refer to members of an array, the comparison is equivalent to comparison of the corresponding subscripts. The address of the first array element is "less than" the address of the last element. In the case of structures, pointers to structure members declared later are "greater than" pointers to members declared earlier in the structure. Pointers to the members of the same union are equal.

- A pointer value can be compared to the constant value 0 for equality (`==`) or inequality (`!=`). A pointer with a value of 0 is called a "null" pointer; that is, it does not point to a valid memory location.
- The equality operators follow the same rules as the relational operators, but permit additional possibilities: a pointer can be compared to a constant integral expression with value 0, or to a pointer to `void`. If two pointers are both null pointers, they compare as equal. Equality operators compare both segment and offset.

Examples

The examples below illustrate relational and equality operators.

```
int x = 0, y = 0;
if ( x < y )
```

Because `x` and `y` are equal, the expression in this example yields the value 0.

```
char array[10];
char *p;

for ( p = array; p < &array[10]; p++ )
    *p = '\0';
```

The fragment in this example sets each element of `array` to a null character constant.

```
enum color { red, white, green } col;
   .
   .
   .
   if ( col == red )
   .
   .
   .
```

These statements declare an enumeration variable named `col` with the tag `color`. At any time, the variable may contain an integer value of 0, 1, or 2, which represents one of the elements of the enumeration set `color`: the color red, white, or green, respectively. If `col` contains 0 when the `if` statement is executed, any statements depending on the `if` will be executed.

See also

Relational Operators: <, >, <=, and >=
Equality Operators: == and !=

C Bitwise Operators

12/22/2021 • 2 minutes to read • Edit Online

The bitwise operators perform bitwise-AND (**&**), bitwise-exclusive-OR (**^**), and bitwise-inclusive-OR (**|**) operations.

Syntax

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*

The operands of bitwise operators must have integral types, but their types can be different. These operators perform the usual arithmetic conversions; the type of the result is the type of the operands after conversion.

The C bitwise operators are described below:

OPERATOR	DESCRIPTION
&	The bitwise-AND operator compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
^	The bitwise-exclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
|	The bitwise-inclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Examples

These declarations are used for the following three examples:

```
short i = 0xAB00;
short j = 0xABCD;
short n;

n = i & j;
```

The result assigned to `n` in this first example is the same as `i` (0xAB00 hexadecimal).

```
n = i | j;

n = i ^ j;
```

The bitwise-inclusive OR in the second example results in the value 0xABCD (hexadecimal), while the bitwise-exclusive OR in the third example produces 0xCD (hexadecimal).

Microsoft Specific

The results of bitwise operation on signed integers is implementation-defined according to the ANSI C standard. For the Microsoft C compiler, bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For example, `-16 & 99` can be expressed in binary as

```
  11111111 11110000
& 00000000 01100011
  _________________
  00000000 01100000
```

The result of the bitwise AND is 96 decimal.

END Microsoft Specific

See also

Bitwise AND Operator: &
Bitwise Exclusive OR Operator: ^
Bitwise Inclusive OR Operator: |

C logical operators

12/22/2021 • 2 minutes to read • Edit Online

The logical operators perform logical-AND (**&&**) and logical-OR (**||**) operations.

Syntax

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

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

Remarks

Logical operators do not perform the usual arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0. The result of a logical operation is either 0 or 1. The result's type is `int`.

The C logical operators are described below:

OPERATOR	DESCRIPTION
&&	The logical-AND operator produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.
||	The logical-OR operator performs an inclusive-OR operation on its operands. The result is 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

The operands of logical-AND and logical-OR expressions are evaluated from left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated. This is called "short-circuit evaluation." There is a sequence point after the first operand. See Sequence Points for more information.

Examples

The following examples illustrate the logical operators:

```
int w, x, y, z;

if ( x < y && y < z )
    printf( "x is less than z\n" );
```

In this example, the **printf** function is called to print a message if `x` is less than `y` and `y` is less than `z`. If `x` is greater than `y`, the second operand (`y < z`) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand has side effects that are being relied on for some other reason.

```
printf( "%d" , (x == w || x == y || x == z) );
```

In this example, if `x` is equal to either `w`, `y`, or `z`, the second argument to the **printf** function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

See also

- Logical AND Operator: &&
- Logical OR Operator: ||

Conditional-Expression Operator

12/22/2021 • 2 minutes to read • Edit Online

C has one ternary operator: the conditional-expression operator (**? :**).

Syntax

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

The *logical-OR-expression* must have integral, floating, or pointer type. It is evaluated in terms of its equivalence to 0. A sequence point follows *logical-OR-expression*. Evaluation of the operands proceeds as follows:

- If *logical-OR-expression* is not equal to 0, *expression* is evaluated. The result of evaluating the expression is given by the nonterminal *expression*. (This means *expression* is evaluated only if *logical-OR-expression* is true.)
- If *logical-OR-expression* equals 0, *conditional-expression* is evaluated. The result of the expression is the value of *conditional-expression*. (This means *conditional-expression* is evaluated only if *logical-OR-expression* is false.)

Note that either *expression* or *conditional-expression* is evaluated, but not both.

The type of the result of a conditional operation depends on the type of the *expression* or *conditional-expression* operand, as follows:

- If *expression* or *conditional-expression* has integral or floating type (their types can be different), the operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.
- If both *expression* and *conditional-expression* have the same structure, union, or pointer type, the type of the result is the same structure, union, or pointer type.
- If both operands have type `void`, the result has type `void`.
- If either operand is a pointer to an object of any type, and the other operand is a pointer to `void`, the pointer to the object is converted to a pointer to `void` and the result is a pointer to `void`.
- If either *expression* or *conditional-expression* is a pointer and the other operand is a constant expression with the value 0, the type of the result is the pointer type.

In the type comparison for pointers, any type qualifiers (`const` or `volatile`) in the type to which the pointer points are insignificant, but the result type inherits the qualifiers from both components of the conditional.

Examples

The following examples show uses of the conditional operator:

```
j = ( i < 0 ) ? ( -i ) : ( i );
```

This example assigns the absolute value of `i` to `j`. If `i` is less than 0, `-i` is assigned to `j`. If `i` is greater than or equal to 0, `i` is assigned to `j`.

```
void f1( void );
void f2( void );
int x;
int y;
    .
    .
    .
( x == y ) ? ( f1() ) : ( f2() );
```

In this example, two functions, `f1` and `f2`, and two variables, `x` and `y`, are declared. Later in the program, if the two variables have the same value, the function `f1` is called. Otherwise, `f2` is called.

See also

Conditional Operator: ? :

C Assignment Operators

12/22/2021 • 2 minutes to read • Edit Online

An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation must be a modifiable l-value. After the assignment, an assignment expression has the value of the left operand but is not an l-value.

Syntax

`assignment-expression` :
  `conditional-expression`
  `unary-expression` `assignment-operator` `assignment-expression`

`assignment-operator` : one of
  `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

The assignment operators in C can both transform and assign values in a single operation. C provides the following assignment operators:

OPERATOR	OPERATION PERFORMED
`=`	Simple assignment
`*=`	Multiplication assignment
`/=`	Division assignment
`%=`	Remainder assignment
`+=`	Addition assignment
`-=`	Subtraction assignment
`<<=`	Left-shift assignment
`>>=`	Right-shift assignment
`&=`	Bitwise-AND assignment
`^=`	Bitwise-exclusive-OR assignment
`|=`	Bitwise-inclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value, and the value is stored in the left operand after the assignment has taken place. The left operand must not be an array, a function, or a constant. The specific conversion path, which depends on the two types, is outlined in detail in Type Conversions.

See also

- Assignment Operators

Simple Assignment (C)

12/22/2021 • 2 minutes to read • Edit Online

The simple-assignment operator assigns its right operand to its left operand. The value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand. The conversion rules for assignment apply (see Assignment Conversions).

```
double x;
int y;

x = y;
```

In this example, the value of `y` is converted to type `double` and assigned to `x`.

See also

C Assignment Operators

C Compound Assignment

12/22/2021 • 2 minutes to read • Edit Online

The compound-assignment operators combine the simple-assignment operator with another binary operator. Compound-assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. For example, a compound-assignment expression such as

> *expression1* += *expression2*

can be understood as

> *expression1* = *expression1* + *expression2*

However, the compound-assignment expression is not equivalent to the expanded version because the compound-assignment expression evaluates *expression1* only once, while the expanded version evaluates *expression1* twice: in the addition operation and in the assignment operation.

The operands of a compound-assignment operator must be of integral or floating type. Each compound-assignment operator performs the conversions that the corresponding binary operator performs and restricts the types of its operands accordingly. The addition-assignment (`+=`) and subtraction-assignment (-=) operators can also have a left operand of pointer type, in which case the right-hand operand must be of integral type. The result of a compound-assignment operation has the value and type of the left operand.

```
#define MASK 0xff00

n &= MASK;
```

In this example, a bitwise-inclusive-AND operation is performed on `n` and `MASK`, and the result is assigned to `n`. The manifest constant `MASK` is defined with a #define preprocessor directive.

See also

C Assignment Operators

Sequential-Evaluation Operator

12/22/2021 • 2 minutes to read • Edit Online

The sequential-evaluation operator, also called the "comma operator," evaluates its two operands sequentially from left to right.

Syntax

expression:
  assignment-expression
  expression **,** *assignment-expression*

The left operand of the sequential-evaluation operator is evaluated as a `void` expression. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The sequential-evaluation operator does not perform type conversions between its operands, and it does not yield an l-value. There is a sequence point after the first operand, which means all side effects from the evaluation of the left operand are completed before beginning evaluation of the right operand. See Sequence Points for more information.

The sequential-evaluation operator is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Commas can be used as separators in some contexts. However, you must be careful not to confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Example

This example illustrates the sequential-evaluation operator:

```
for ( i = j = 1; i + j < 20; i += i, j-- );
```

In this example, each operand of the `for` statement's third expression is evaluated independently. The left operand `i += i` is evaluated first; then the right operand, `j--`, is evaluated.

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

In the function call to `func_one`, three arguments, separated by commas, are passed: `x`, `y + 2`, and `z`. In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

See also

Comma Operator: ,

Type Conversions (C)

12/22/2021 • 2 minutes to read • Edit Online

Type conversions depend on the specified operator and the type of the operand or operators. Type conversions are performed in the following cases:

- When a value of one type is assigned to a variable of a different type or an operator converts the type of its operand or operands before performing an operation
- When a value of one type is explicitly cast to a different type
- When a value is passed as an argument to a function or when a type is returned from a function

A character, a short integer, or an integer bit field, all either signed or not, or an object of enumeration type, can be used in an expression wherever an integer can be used. If an `int` can represent all the values of the original type, then the value is converted to `int`; otherwise, it is converted to `unsigned int`. This process is called "integral promotion." Integral promotions preserve value. That is, the value after promotion is guaranteed to be the same as before the promotion. See Usual Arithmetic Conversions for more information.

See also

Expressions and Assignments

Assignment conversions

12/22/2021 • 2 minutes to read • Edit Online

In assignment operations, the type of the value being assigned is converted to the type of the variable that receives the assignment. C allows conversions by assignment between integral and floating types, even if information is lost in the conversion. The conversion method used depends on the types involved in the assignment, as described in Usual Arithmetic Conversions and in the following sections:

- Conversions from signed integral types
- Conversions from unsigned integral types
- Conversions from floating-point types
- Conversions to and from pointer types
- Conversions from other types

Type qualifiers do not affect the allowability of the conversion although a `const` l-value cannot be used on the left side of the assignment.

See also

Type conversions

Conversions from signed integral types

12/22/2021 • 3 minutes to read • Edit Online

When a signed integer is converted to an integer or a floating-point type, if the original value is representable in the result type, the value is unchanged.

When a signed integer is converted to an integer of greater size, the value is sign-extended. When converted to an integer of smaller size, the high-order bits are truncated. The result is interpreted using the result type, as shown in this example:

```
int i = -3;
unsigned short u;

u = i;
printf_s( "%hu\n", u );  // Prints 65533
```

When converting a signed integer to a floating-point type, if the original value isn't representable exactly in the result type, the result is the next higher or lower representable value.

For information about the sizes of integral and floating-point types, see Storage of basic types.

The following table summarizes conversions from signed integral types. It assumes the `char` type is signed by default. If you use a compile-time option to change the default for the `char` type to unsigned, the conversions given in the Conversions from unsigned integral types table for the `unsigned char` type apply, instead of the conversions in this table.

Microsoft Specific

In the Microsoft compiler, `int` and `long` are distinct but equivalent types. Conversion of an `int` value proceeds in the same way as conversion of a `long`.

END Microsoft Specific

Table of conversions from signed integral types

FROM	TO	METHOD
`char` [1]	`short`	Sign-extend
`char`	`long`	Sign-extend
`char`	`long long`	Sign-extend
`char`	`unsigned char`	Preserve pattern; high-order bit loses function as sign bit
`char`	`unsigned short`	Sign-extend to `short`; convert `short` to `unsigned short`
`char`	`unsigned long`	Sign-extend to `long`; convert `long` to `unsigned long`

FROM	TO	METHOD
`char`	`unsigned long long`	Sign-extend to `long long`; convert `long long` to `unsigned long long`
`char`	`float`	Sign-extend to `long`; convert `long` to `float`
`char`	`double`	Sign-extend to `long`; convert `long` to `double`
`char`	`long double`	Sign-extend to `long`; convert `long` to `double`
`short`	`char`	Preserve low-order byte
`short`	`long`	Sign-extend
`short`	`long long`	Sign-extend
`short`	`unsigned char`	Preserve low-order byte
`short`	`unsigned short`	Preserve bit pattern; high-order bit loses function as sign bit
`short`	`unsigned long`	Sign-extend to `long`; convert `long` to `unsigned long`
`short`	`unsigned long long`	Sign-extend to `long long`; convert `long long` to `unsigned long long`
`short`	`float`	Sign-extend to `long`; convert `long` to `float`
`short`	`double`	Sign-extend to `long`; convert `long` to `double`
`short`	`long double`	Sign-extend to `long`; convert `long` to `double`
`long`	`char`	Preserve low-order byte
`long`	`short`	Preserve low-order word
`long`	`long long`	Sign-extend
`long`	`unsigned char`	Preserve low-order byte
`long`	`unsigned short`	Preserve low-order word
`long`	`unsigned long`	Preserve bit pattern; high-order bit loses function as sign bit

FROM	TO	METHOD
`long`	`unsigned long long`	Sign-extend to `long long`; convert `long long` to `unsigned long long`
`long`	`float`	Represent as `float`. If `long` can't be represented exactly, some precision is lost.
`long`	`double`	Represent as `double`. If `long` can't be represented exactly as a `double`, some precision is lost.
`long`	`long double`	Represent as `double`. If `long` can't be represented exactly as a `double`, some precision is lost.
`long long`	`char`	Preserve low-order byte
`long long`	`short`	Preserve low-order word
`long long`	`long`	Preserve low-order dword
`long long`	`unsigned char`	Preserve low-order byte
`long long`	`unsigned short`	Preserve low-order word
`long long`	`unsigned long`	Preserve low-order dword
`long long`	`unsigned long long`	Preserve bit pattern; high-order bit loses function as sign bit
`long long`	`float`	Represent as `float`. If `long long` can't be represented exactly, some precision is lost.
`long long`	`double`	Represent as `double`. If `long long` can't be represented exactly as a `double`, some precision is lost.
`long long`	`long double`	Represent as `double`. If `long long` can't be represented exactly as a `double`, some precision is lost.

[1] All `char` entries assume that the `char` type is signed by default.

See also

Assignment conversions

Conversions from unsigned integral types

12/22/2021 • 2 minutes to read • Edit Online

When an unsigned integer is converted to an integer or floating-point type, if the original value is representable in the result type the value is unchanged.

When converting an unsigned integer to an integer of greater size, the value is zero-extended. When converting to an integer of smaller size, the high-order bits are truncated. The result is interpreted using the result type, as shown in this example.

```
unsigned k = 65533;
short j;

j = k;
printf_s( "%hd\n", j );   // Prints -3
```

When converting an unsigned integer to a floating-point type, if the original value can't be represented exactly in the result type, the result is the next higher or lower representable value.

See Storage of basic types for information about the sizes of integral and floating-point types.

Microsoft Specific

In the Microsoft compiler, `unsigned` (or `unsigned int`) and `unsigned long` are distinct but equivalent types. Conversion of an `unsigned int` value proceeds in the same way as conversion of an `unsigned long`.

END Microsoft Specific

The following table summarizes conversions from unsigned integral types.

Table of conversions from unsigned integral types

FROM	TO	METHOD
`unsigned char`	`char`	Preserve bit pattern; high-order bit becomes sign bit
`unsigned char`	`short`	Zero-extend
`unsigned char`	`long`	Zero-extend
`unsigned char`	`long long`	Zero-extend
`unsigned char`	`unsigned short`	Zero-extend
`unsigned char`	`unsigned long`	Zero-extend
`unsigned char`	`unsigned long long`	Zero-extend
`unsigned char`	`float`	Convert to `long`; convert `long` to `float`

FROM	TO	METHOD
`unsigned char`	`double`	Convert to `long`; convert `long` to `double`
`unsigned char`	`long double`	Convert to `long`; convert `long` to `double`
`unsigned short`	`char`	Preserve low-order byte
`unsigned short`	`short`	Preserve bit pattern; high-order bit becomes sign bit
`unsigned short`	`long`	Zero-extend
`unsigned short`	`long long`	Zero-extend
`unsigned short`	`unsigned char`	Preserve low-order byte
`unsigned short`	`unsigned long`	Zero-extend
`unsigned short`	`unsigned long long`	Zero-extend
`unsigned short`	`float`	Convert to `long`; convert `long` to `float`
`unsigned short`	`double`	Convert to `long`; convert `long` to `double`
`unsigned short`	`long double`	Convert to `long`; convert `long` to `double`
`unsigned long`	`char`	Preserve low-order byte
`unsigned long`	`short`	Preserve low-order word
`unsigned long`	`long`	Preserve bit pattern; high-order bit becomes sign bit
`unsigned long`	`long long`	Zero-extend
`unsigned long`	`unsigned char`	Preserve low-order byte
`unsigned long`	`unsigned short`	Preserve low-order word
`unsigned long`	`unsigned long long`	Zero-extend
`unsigned long`	`float`	Convert to `long`; convert `long` to `float`
`unsigned long`	`double`	Convert directly to `double`

FROM	TO	METHOD
`unsigned long`	`long double`	Convert to `long`; convert `long` to `double`
`unsigned long long`	`char`	Preserve low-order byte
`unsigned long long`	`short`	Preserve low-order word
`unsigned long long`	`long`	Preserve low-order dword
`unsigned long long`	`long long`	Preserve bit pattern; high-order bit becomes sign bit
`unsigned long long`	`unsigned char`	Preserve low-order byte
`unsigned long long`	`unsigned short`	Preserve low-order word
`unsigned long long`	`unsigned long`	Preserve low-order dword
`unsigned long long`	`float`	Convert to `long`; convert `long` to `float`
`unsigned long long`	`double`	Convert directly to `double`
`unsigned long long`	`long double`	Convert to `long`; convert `long` to `double`

See also

Assignment conversions

Conversions from floating-point types

12/22/2021 • 4 minutes to read • Edit Online

A floating-point value that's converted to another floating-point type undergoes no change in value if the original value is representable exactly in the result type. If the original value is numeric but isn't representable exactly, the result is either the next greater or next lower representable value. See Limits on floating-point constants for the range of floating-point types.

A floating-point value that is converted to an integral type is first truncated by discarding any fractional value. If this truncated value is representable in the result type, the result must be that value. When it isn't representable, the result value is undefined.

Microsoft Specific

Microsoft compilers use IEEE-754 binary32 representation for `float` values, and binary64 representation for `long double` and `double`. Since `long double` and `double` use the same representation, they have the same range and precision.

When the compiler converts a `double` or `long double` floating-point number to a `float`, it rounds the result according to the floating-point environment controls, which default to "round to nearest, ties to even." If a numeric value is too high or too low to be represented as a numeric `float` value, the conversion result is positive or negative infinity according to the sign of the original value, and an overflow exception is raised, if enabled.

When converting to integer types, the result of a conversion to a type smaller than `long` is the result of converting the value to `long`, and then converting to the result type.

For conversion to integer types at least as large as `long`, a conversion of a value that is too high or too low to represent in the result type may return any of the following values:

- The result may be a *sentinel value*, which is the representable value farthest from zero. For signed types, it's the lowest representable value (0x800...0). For unsigned types, it's the highest representable value (0xFF...F).
- The result may be *saturated*, where values too high to represent are converted to the highest representable value, and values too low to represent are converted to the lowest representable value. One of these two values is also used as the sentinel value.
- For conversion to `unsigned long` or `unsigned long long`, the result of converting an out-of-range value may be some value other than the highest or lowest representable value. Whether the result is a sentinel or saturated value or not depends on the compiler options and target architecture. Future compiler releases may return a saturated or sentinel value instead.

END Microsoft Specific

The following table summarizes conversions from floating types.

Table of conversions from floating-point types

FROM	TO	METHOD

FROM	TO	METHOD
`float`	`char`	Convert to `long`; convert `long` to `char`
`float`	`short`	Convert to `long`; convert `long` to `short`
`float`	`int`	Truncate at decimal point. If result is too large to be represented as `int`, result is undefined.
`float`	`long`	Truncate at decimal point. If result is too large to be represented as `long`, result is undefined.
`float`	`long long`	Truncate at decimal point. If result is too large to be represented as `long long`, result is undefined.
`float`	`unsigned char`	Convert to `long`; convert `long` to `unsigned char`
`float`	`unsigned short`	Convert to `long`; convert `long` to `unsigned short`
`float`	`unsigned`	Truncate at decimal point. If result is too large to be represented as `unsigned`, result is undefined.
`float`	`unsigned long`	Truncate at decimal point. If result is too large to be represented as `unsigned long`, result is undefined.
`float`	`unsigned long long`	Truncate at decimal point. If result is too large to be represented as `unsigned long long`, result is undefined.
`float`	`double`	Represent as a `double`.
`float`	`long double`	Represent as a `long double`.
`double`	`char`	Convert to `float`; convert `float` to `char`
`double`	`short`	Convert to `float`; convert `float` to `short`
`double`	`int`	Truncate at decimal point. If result is too large to be represented as `int`, result is undefined.

FROM	TO	METHOD
`double`	`long`	Truncate at decimal point. If result is too large to be represented as `long`, result is undefined.
`double`	`unsigned char`	Convert to `long`; convert `long` to `unsigned char`
`double`	`unsigned short`	Convert to `long`; convert `long` to `unsigned short`
`double`	`unsigned`	Truncate at decimal point. If result is too large to be represented as `unsigned`, result is undefined.
`double`	`unsigned long`	Truncate at decimal point. If result is too large to be represented as `unsigned long`, result is undefined.
`double`	`unsigned long long`	Truncate at decimal point. If result is too large to be represented as `unsigned long long`, result is undefined.
`double`	`float`	Represent as a `float`. If `double` value can't be represented exactly as `float`, loss of precision occurs. If value is too large to be represented as `float`, the result is undefined.
`double`	`long double`	The `long double` value is treated as `double`.

Conversions from `long double` follow the same method as conversions from `double`.

See also

Assignment conversions

Conversions to and from Pointer Types

12/22/2021 • 2 minutes to read • Edit Online

A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage. A pointer to an object can be converted to a pointer to an object whose type requires less or equally strict storage alignment, and back again without change.

A pointer to `void` can be converted to or from a pointer to any type, without restriction or loss of information. If the result is converted back to the original type, the original pointer is recovered.

If a pointer is converted to another pointer with the same type but having different or additional qualifiers, the new pointer is the same as the old except for restrictions imposed by the new qualifier.

A pointer value can also be converted to an integral value. The conversion path depends on the size of the pointer and the size of the integral type, according to the following rules:

- If the size of the pointer is greater than or equal to the size of the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type.

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).
- If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in the tables Conversion from Signed Integral Types and Conversion from Unsigned Integral Types. It is then treated as a pointer value.

An integral constant expression with value 0 or such an expression cast to type `void` * can be converted by a type cast, by assignment, or by comparison to a pointer of any type. This produces a null pointer that is equal to another null pointer of the same type, but this null pointer is not equal to any pointer to a function or to an object. Integers other than the constant 0 can be converted to pointer type, but the result is not portable.

See also

Assignment Conversions

Conversions from other types

12/22/2021 • 2 minutes to read • Edit Online

Since an `enum` value is an `int` value by definition, conversions to and from an `enum` value are the same as those for the `int` type. For the Microsoft C compiler, an integer is the same as a `long`.

Microsoft Specific

No conversions between structure or union types are allowed.

Any value can be converted to type `void`, but the result of such a conversion can be used only in a context where an expression value is discarded, such as in an expression statement.

The `void` type has no value, by definition. Therefore, it cannot be converted to any other type, and other types cannot be converted to `void` by assignment. However, you can explicitly cast a value to type `void`, as discussed in Type-Cast Conversions.

END Microsoft Specific

See also

Assignment Conversions

Type-Cast Conversions

12/22/2021 • 2 minutes to read • Edit Online

You can use type casts to explicitly convert types.

Syntax

cast-expression:
  unary expression
  (*type-name* **)** *cast-expression*

type-name:
  specifier-qualifier-list abstract-declarator$_{\text{opt}}$

The *type-name* is a type and *cast-expression* is a value to be converted to that type. An expression with a type cast is not an l-value. The *cast-expression* is converted as though it had been assigned to a variable of type *type-name*. The conversion rules for assignments (outlined in Assignment Conversions) apply to type casts as well. The following table shows the types that can be cast to any given type.

Legal Type Casts

DESTINATION TYPES	POTENTIAL SOURCES
Integral types	Any integer type or floating-point type, or pointer to an object
Floating-point	Any arithmetic type
A pointer to an object, or (`void` *)	Any integer type, (`void` *), a pointer to an object, or a function pointer
Function pointer	Any integral type, a pointer to an object, or a function pointer
A structure, union, or array	None
Void type	Any type

Any identifier can be cast to `void` type. However, if the type specified in a type-cast expression is not `void`, then the identifier being cast to that type cannot be a `void` expression. Any expression can be cast to `void`, but an expression of type `void` cannot be cast to any other type. For example, a function with `void` return type cannot have its return cast to another type.

Note that a `void` * expression has a type pointer to `void`, not type `void`. If an object is cast to `void` type, the resulting expression cannot be assigned to any item. Similarly, a type-cast object is not an acceptable l-value, so no assignment can be made to a type-cast object.

Microsoft Specific

A type cast can be an l-value expression as long as the size of the identifier does not change. For information on l-value expressions, see L-Value and R-Value Expressions.

END Microsoft Specific

You can convert an expression to type `void` with a cast, but the resulting expression can be used only where a value is not required. An object pointer converted to `void` * and back to the original type will return to its original value.

See also

Type Conversions

Function-Call Conversions

12/22/2021 • 2 minutes to read • Edit Online

The type of conversion performed on the arguments in a function call depends on the presence of a function prototype (forward declaration) with declared argument types for the called function.

If a function prototype is present and includes declared argument types, the compiler performs type checking (see Functions).

If no function prototype is present, only the usual arithmetic conversions are performed on the arguments in the function call. These conversions are performed independently on each argument in the call. This means that a `float` value is converted to a `double`; a `char` or `short` value is converted to an `int`; and an `unsigned char` or `unsigned short` is converted to an `unsigned int`.

See also

Type Conversions

Statements (C)

12/22/2021 • 2 minutes to read • Edit Online

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. Following a brief overview of statement syntax, this section describes the C statements in alphabetical order:

break statement
compound statement
continue statement
do-while statement
expression statement

for statement
goto and labeled statements
if statement
null statement
return statement

switch statement
try-except statement
try-finally statement
while statement

See also

C Language Reference

Overview of C Statements

12/22/2021 • 2 minutes to read • Edit Online

C statements consist of tokens, expressions, and other statements. A statement that forms a component of another statement is called the "body" of the enclosing statement. Each statement type given by the following syntax is discussed in this section.

Syntax

statement: labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

try-except-statement /* Microsoft-specific */

try-finally-statement /* Microsoft-specific */

Frequently the statement body is a "compound statement." A compound statement consists of other statements that can include keywords. The compound statement is delimited by braces (**{ }**). All other C statements end with a semicolon (**;**). The semicolon is a statement terminator.

The expression statement contains a C expression that can contain the arithmetic or logical operators introduced in Expressions and Assignments. The null statement is an empty statement.

Any C statement can begin with an identifying label consisting of a name and a colon. Since only the `goto` statement recognizes statement labels, statement labels are discussed with `goto`. See The goto and Labeled Statements for more information.

See also

Statements

break Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

The `break` statement terminates the execution of the nearest enclosing `do`, `for`, `switch`, or `while` statement in which it appears. Control passes to the statement that follows the terminated statement.

Syntax

jump-statement:
  break ;

The `break` statement is frequently used to terminate the processing of a particular case within a `switch` statement. Lack of an enclosing iterative or `switch` statement generates an error.

Within nested statements, the `break` statement terminates only the `do`, `for`, `switch`, or `while` statement that immediately encloses it. You can use a `return` or `goto` statement to transfer control elsewhere out of the nested structure.

This example illustrates the `break` statement:

```
#include <stdio.h>
int main() {
   char c;
   for(;;) {
      printf_s( "\nPress any key, Q to quit: " );

      // Convert to character value
      scanf_s("%c", &c);
      if (c == 'Q')
          break;
   }
} // Loop exits only when 'Q' is pressed
```

See also

break Statement

Compound Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

A compound statement (also called a "block") typically appears as the body of another statement, such as the `if` statement. Declarations and Types describes the form and meaning of the declarations that can appear at the head of a compound statement.

Syntax

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

If there are declarations, they must come before any statements. The scope of each identifier declared at the beginning of a compound statement extends from its declaration point to the end of the block. It is visible throughout the block unless a declaration of the same identifier exists in an inner block.

Identifiers in a compound statement are presumed `auto` unless explicitly declared otherwise with `register`, `static`, or `extern`, except functions, which can only be `extern`. You can leave off the `extern` specifier in function declarations and the function will still be `extern`.

Storage is not allocated and initialization is not permitted if a variable or function is declared in a compound statement with storage class `extern`. The declaration refers to an external variable or function defined elsewhere.

Variables declared in a block with the `auto` or `register` keyword are reallocated and, if necessary, initialized each time the compound statement is entered. These variables are not defined after the compound statement is exited. If a variable declared inside a block has the `static` attribute, the variable is initialized when program execution begins and keeps its value throughout the program. See Storage Classes for information about `static`.

This example illustrates a compound statement:

```
if ( i > 0 )
{
    line[i] = x;
    x++;
    i--;
}
```

In this example, if `i` is greater than 0, all statements inside the compound statement are executed in order.

See also

Statements

`continue` statement (C)

12/22/2021 • 2 minutes to read • Edit Online

The `continue` statement passes control to the next iteration of the nearest enclosing `do`, `for`, or `while` statement in which it appears, bypassing any remaining statements in the `do`, `for`, or `while` statement body.

Syntax

`jump-statement`:
  `continue ;`

The next iteration of a `do`, `for`, or `while` statement is determined as follows:

- Within a `do` or a `while` statement, the next iteration starts by reevaluating the expression of the `do` or `while` statement.
- A `continue` statement in a `for` statement causes evaluation of the loop expression of the `for` statement. Then the code reevaluates the conditional expression. Depending on the result, it either terminates or iterates the statement body. For more information on the `for` statement and its nonterminals, see The `for` statement.

Here's an example of the `continue` statement:

```
while ( i-- > 0 )
{
    x = f( i );
    if ( x == 1 )
        continue;
    y += x * x;
}
```

In this example, the statement body is executed while `i` is greater than 0. First `f(i)` is assigned to `x`; then, if `x` is equal to 1, the `continue` statement is executed. The rest of the statements in the body get ignored. Execution resumes at the top of the loop with the evaluation of the loop's test.

See also

`continue` statement (C++)

do-while Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

The *do-while* statement lets you repeat a statement or compound statement until a specified expression becomes false.

Syntax

iteration-statement: `do` *statement* **while (** *expression* **) ;**

The *expression* in a *do-while* statement is evaluated after the body of the loop is executed. Therefore, the body of the loop is always executed at least once.

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The statement body is executed.

2. Next, *expression* is evaluated. If *expression* is false, the *do-while* statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

The *do-while* statement can also terminate when a `break`, `goto`, or `return` statement is executed within the statement body.

This is an example of the *do-while* statement:

```
do
{
    y = f( x );
    x--;
} while ( x > 0 );
```

In this *do-while* statement, the two statements `y = f( x );` and `x--;` are executed, regardless of the initial value of `x`. Then `x > 0` is evaluated. If `x` is greater than 0, the statement body is executed again and `x > 0` is reevaluated. The statement body is executed repeatedly as long as `x` remains greater than 0. Execution of the *do-while* statement terminates when `x` becomes 0 or negative. The body of the loop is executed at least once.

See also

do-while Statement (C++)

Expression Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

When an expression statement is executed, the expression is evaluated according to the rules outlined in Expressions and Assignments.

Syntax

expression-statement:
  expression$_{\text{opt}}$ **;**

All side effects from the expression evaluation are completed before the next statement is executed. An empty expression statement is called a null statement. See The Null Statement for more information.

These examples demonstrate expression statements.

```
x = ( y + 3 );            /* x is assigned the value of y + 3  */
x++;                      /* x is incremented                  */
x = y = 0;                /* Both x and y are initialized to 0 */
proc( arg1, arg2 );       /* Function call returning void      */
y = z = ( f( x ) + 3 );   /* A function-call expression        */
```

In the last statement, the function-call expression, the value of the expression, which includes any value returned by the function, is increased by 3 and then assigned to both the variables `y` and `z`.

See also

Statements

for Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

The `for` statement lets you repeat a statement or compound statement a specified number of times. The body of a `for` statement is executed zero or more times until an optional condition becomes false. You can use optional expressions within the `for` statement to initialize and change values during the `for` statement's execution.

Syntax

iteration-statement:
  `for` (*init-expression*$_{opt}$; *cond-expression*$_{opt}$; *loop-expression*$_{opt}$) *statement*

Execution of a `for` statement proceeds as follows:

1. The *init-expression*, if any, is evaluated. This specifies the initialization for the loop. There is no restriction on the type of *init-expression*.

2. The *cond-expression*, if any, is evaluated. This expression must have arithmetic or pointer type. It is evaluated before each iteration. Three results are possible:

   - If *cond-expression* is `true` (nonzero), *statement* is executed; then *loop-expression*, if any, is evaluated. The *loop-expression* is evaluated after each iteration. There is no restriction on its type. Side effects will execute in order. The process then begins again with the evaluation of *cond-expression*.

   - If *cond-expression* is omitted, *cond-expression* is considered true, and execution proceeds exactly as described in the previous paragraph. A `for` statement without a *cond-expression* argument terminates only when a `break` or `return` statement within the statement body is executed, or when a `goto` (to a labeled statement outside the `for` statement body) is executed.

   - If *cond-expression* is `false` (0), execution of the `for` statement terminates and control passes to the next statement in the program.

A `for` statement also terminates when a `break`, `goto`, or `return` statement within the statement body is executed. A `continue` statement in a `for` loop causes *loop-expression* to be evaluated. When a `break` statement is executed inside a `for` loop, *loop-expression* is not evaluated or executed. This statement

```
for( ; ; )
```

is the customary way to produce an infinite loop which can only be exited with a `break`, `goto`, or `return` statement.

Example

This example illustrates the `for` statement:

```c
// c_for.c
int main()
{
   char* line = "H e  \tl\tlo World\0";
   int space = 0;
   int tab = 0;
   int i;
   int max = strlen(line);
   for (i = 0; i < max; i++ )
   {
      if ( line[i] == ' ' )
      {
          space++;
      }
      if ( line[i] == '\t' )
      {
          tab++;
      }
   }

   printf("Number of spaces: %i\n", space);
   printf("Number of tabs: %i\n", tab);
   return 0;
}
```

Output

```
Number of spaces: 4
Number of tabs: 2
```

See also

Statements

goto and Labeled Statements (C)

12/22/2021 • 2 minutes to read • Edit Online

The `goto` statement transfers control to a label. The given label must reside in the same function and can appear before only one statement in the same function.

Syntax

statement:
  labeled-statement
  jump-statement

jump-statement:
  `goto` *identifier* ;

labeled-statement:
  identifier : *statement*

A statement label is meaningful only to a `goto` statement; in any other context, a labeled statement is executed without regard to the label.

A *jump-statement* must reside in the same function and can appear before only one statement in the same function. The set of *identifier* names following a `goto` has its own name space so the names do not interfere with other identifiers. Labels cannot be redeclared. See Name Spaces for more information.

It is good programming style to use the `break`, `continue`, and `return` statement in preference to `goto` whenever possible. Since the `break` statement only exits from one level of the loop, a `goto` may be necessary for exiting a loop from within a deeply nested loop.

This example demonstrates the `goto` statement:

```
// goto.c
#include <stdio.h>

int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }

    /* This message does not print: */
    printf_s( "Loop exited. i = %d\n", i );

    stop: printf_s( "Jumped to stop. i = %d\n", i );
}
```

In this example, a `goto` statement transfers control to the point labeled `stop` when `i` equals 5.

See also

Statements

if Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

The `if` statement controls conditional branching. The body of an `if` statement is executed if the value of the expression is nonzero. The syntax for the `if` statement has two forms.

Syntax

selection-statement: **if (** *expression* **)** *statement*

if (*expression* **)** *statement* `else` *statement*

In both forms of the `if` statement, the expressions, which can have any value except a structure, are evaluated, including all side effects.

In the first form of the syntax, if *expression* is true (nonzero), *statement* is executed. If *expression* is false, *statement* is ignored. In the second form of syntax, which uses `else`, the second *statement* is executed if *expression* is false. With both forms, control then passes from the `if` statement to the next statement in the program unless one of the statements contains a `break`, `continue`, or `goto`.

The following are examples of the `if` statement:

```
if ( i > 0 )
    y = x / i;
else
{
    x = i;
    y = f( x );
}
```

In this example, the statement `y = x/i;` is executed if `i` is greater than 0. If `i` is less than or equal to 0, `i` is assigned to `x` and `f( x )` is assigned to `y`. Note that the statement forming the `if` clause ends with a semicolon.

When nesting `if` statements and `else` clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the compiler resolves ambiguities by associating each `else` with the closest `if` that lacks an `else`.

```
if ( i > 0 )           /* Without braces */
    if ( j > i )
        x = j;
    else
        x = i;
```

The `else` clause is associated with the inner `if` statement in this example. If `i` is less than or equal to 0, no value is assigned to `x`.

```
if ( i > 0 )
{                      /* With braces */
    if ( j > i )
        x = j;
}
else
    x = i;
```

The braces surrounding the inner `if` statement in this example make the `else` clause part of the outer `if` statement. If `i` is less than or equal to 0, `i` is assigned to `x`.

See also

if-else Statement (C++)

Null Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

A "null statement" is a statement containing only a semicolon; it can appear wherever a statement is expected. Nothing happens when a null statement is executed. The correct way to code a null statement is:

Syntax

```
;
```

Remarks

Statements such as `do`, `for`, `if`, and `while` require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

As with any other C statement, you can include a label before a null statement. To label an item that is not a statement, such as the closing brace of a compound statement, you can label a null statement and insert it immediately before the item to get the same effect.

This example illustrates the null statement:

```
for ( i = 0; i < 10; line[i++] = 0 )
     ;
```

In this example, the loop expression of the `for` statement `line[i++] = 0` initializes the first 10 elements of `line` to 0. The statement body is a null statement, since no further statements are necessary.

See also

Statements

return Statement (C)

12/22/2021 • 4 minutes to read • Edit Online

A `return` statement ends the execution of a function, and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A `return` statement can return a value to the calling function. For more information, see Return type.

Syntax

> *jump-statement*:
> `return` *expression*$_{\text{opt}}$ `;`

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. The expression, if present, is evaluated and then converted to the type returned by the function. When a `return` statement contains an expression in functions that have a `void` return type, the compiler generates a warning, and the expression isn't evaluated.

If no `return` statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. In this case, the return value of the called function is undefined. If the function has a return type other than `void`, it's a serious bug, and the compiler prints a warning diagnostic message. If the function has a `void` return type, this behavior is okay, but may be considered poor style. Use a plain `return` statement to make your intent clear.

As a good engineering practice, always specify a return type for your functions. If a return value isn't required, declare the function to have `void` return type. If a return type isn't specified, the C compiler assumes a default return type of `int`.

Many programmers use parentheses to enclose the *expression* argument of the `return` statement. However, C doesn't require the parentheses.

The compiler may issue a warning diagnostic message about unreachable code if it finds any statements placed after the `return` statement.

In a `main` function, the `return` statement and expression are optional. What happens to the returned value, if one is specified, depends on the implementation. **Microsoft-specific**: The Microsoft C implementation returns the expression value to the process that invoked the program, such as *`cmd.exe`*. If no `return` expression is supplied, the Microsoft C runtime returns a value that indicates success (0) or failure (a non-zero value).

Example

This example is one program in several parts. It demonstrates the `return` statement, and how it's used both to end function execution, and optionally, to return a value.

```c
// C_return_statement.c
// Compile using: cl /W4 C_return_statement.c
#include <limits.h>      // for INT_MAX
#include <stdio.h>       // for printf

long long square( int value )
{
    // Cast one operand to long long to force the
    // expression to be evaluated as type long long.
    // Note that parentheses around the return expression
    // are allowed, but not required here.
    return ( value * (long long) value );
}
```

The `square` function returns the square of its argument, in a wider type to prevent an arithmetic error. **Microsoft-specific**: In the Microsoft C implementation, the `long long` type is large enough to hold the product of two `int` values without overflow.

The parentheses around the `return` expression in `square` are evaluated as part of the expression, and aren't required by the `return` statement.

```c
double ratio( int numerator, int denominator )
{
    // Cast one operand to double to force floating-point
    // division. Otherwise, integer division is used,
    // then the result is converted to the return type.
    return numerator / (double) denominator;
}
```

The `ratio` function returns the ratio of its two `int` arguments as a floating-point `double` value. The `return` expression is forced to use a floating-point operation by casting one of the operands to `double`. Otherwise, the integer division operator would be used, and the fractional part would be lost.

```c
void report_square( void )
{
    int value = INT_MAX;
    long long squared = 0LL;
    squared = square( value );
    printf( "value = %d, squared = %lld\n", value, squared );
    return; // Use an empty expression to return void.
}
```

The `report_square` function calls `square` with a parameter value of `INT_MAX`, the largest signed integer value that fits in an `int`. The `long long` result is stored in `squared`, then printed. The `report_square` function has a `void` return type, so it doesn't have an expression in its `return` statement.

```c
void report_ratio( int top, int bottom )
{
    double fraction = ratio( top, bottom );
    printf( "%d / %d = %.16f\n", top, bottom, fraction );
    // It's okay to have no return statement for functions
    // that have void return types.
}
```

The `report_ratio` function calls `ratio` with parameter values of `1` and `INT_MAX`. The `double` result is stored in `fraction`, then printed. The `report_ratio` function has a `void` return type, so it doesn't need to explicitly return a value. Execution of `report_ratio` "falls off the bottom" and returns no value to the caller.

```
int main()
{
    int n = 1;
    int x = INT_MAX;

    report_square();
    report_ratio( n, x );

    return 0;
}
```

The `main` function calls two functions: `report_square` and `report_ratio`. As `report_square` takes no parameters and returns `void`, we don't assign its result to a variable. Likewise, `report_ratio` returns `void`, so we don't save its return value, either. After each of these function calls, execution continues at the next statement. Then `main` returns a value of `0` (typically used to report success) to end the program.

To compile the example, create a source code file named *`C_return_statement.c`*. Then, copy all the example code, in the order shown. Save the file, and compile it in a Developer command prompt window by using the command:

`cl /W4 C_return_statement.c`

Then, to run the example code, enter `C_return_statement.exe` at the command prompt. The output of the example looks like this:

```
value = 2147483647, squared = 4611686014132420609
1 / 2147483647 = 0.0000000004656613
```

See also

Statements

`_Static_assert` keyword and `static_assert` macro (C11)

12/22/2021 • 2 minutes to read • Edit Online

Tests an assertion at compile time. If the specified constant expression is `false`, the compiler displays the specified message and the compilation fails with error C2338; otherwise, there's no effect. New in C11.

`_Static_assert` is a keyword introduced in C11. `static_assert` is a macro, introduced in C11, that maps to the `_Static_assert` keyword.

Syntax

```
_Static_assert(constant-expression, string-literal);
static_assert(constant-expression, string-literal);
```

Parameters

`constant-expression`
An integral constant expression that can be evaluated at compile time. If the expression is zero (false), displays the *`string-literal`* parameter and the compilation fails with an error. If the expression is nonzero (true), then there's no effect.

`string-literal`
The message displayed if *`constant-expression`* evaluates to zero (false). The message must be made using the base character set of the compiler. The characters can't be multibyte or wide characters.

Remarks

The `_Static_assert` keyword, and the `static_assert` macro, both test a software assertion at compile time. They can be used at global or function scope.

In contrast, the `assert` macro and `_assert` and `_wassert` functions test a software assertion at runtime and incur a runtime cost.

Microsoft-specific behavior

In C, when you don't include `<assert.h>`, the Microsoft compiler treats `static_assert` as a keyword that maps to `_Static_assert`. Using `static_assert` is preferred because the same code will work in both C and C++.

Example of a compile-time assert

In the following example, `static_assert` and `_Static_assert` are used to verify how many elements are in an enum and that integers are 32 bits wide.

```c
// requires /std:c11 or higher
#include <assert.h>

enum Items
{
    A,
    B,
    C,
    LENGTH
};

int main()
{
    // _Static_assert is a C11 keyword
    _Static_assert(LENGTH == 3, "Expected Items enum to have three elements");

    // Preferred: static_assert maps to _Static_assert and is compatible with C++
    static_assert(sizeof(int) == 4, "Expecting 32 bit integers");

    return 0;
}
```

Requirements

MACRO	REQUIRED HEADER
`static_assert`	<assert.h>

Compile with `/std:c11`.

Windows SDK 10.0.20348.0 (version 2104) or later. For more information on installing the Windows SDK for C11 and C17 development, see Install C11 and C17 support in Visual Studio.

See also

`_STATIC_ASSERT` Macro
`assert` macro and `_assert` and `_wassert` functions `/std` (Specify language standard version)

`switch` Statement (C)

12/22/2021 • 3 minutes to read • Edit Online

The `switch` and `case` statements help control complex conditional and branching operations. The `switch` statement transfers control to a statement within its body.

Syntax

`selection-statement`:
  `switch (` *`expression`* `)` *`statement`*

`labeled-statement`:
  `case` *`constant-expression`* `:` *`statement`*
  `default` `:` *`statement`*

Remarks

A `switch` statement causes control to transfer to one *`labeled-statement`* in its statement body, depending on the value of *`expression`*.

The values of *`expression`* and each *`constant-expression`* must have an integral type. A *`constant-expression`* must have an unambiguous constant integral value at compile time.

Control passes to the `case` statement whose *`constant-expression`* value matches the value of *`expression`*. The `switch` statement can include any number of `case` instances. However, no two *`constant-expression`* values within the same `switch` statement can have the same value. Execution of the `switch` statement body begins at the first statement in or after the matching *`labeled-statement`*. Execution proceeds until the end of the body, or until a `break` statement transfers control out of the body.

Use of the `switch` statement usually looks something like this:

```
switch ( expression )
{
    // declarations
    // . . .
    case constant_expression:
        // statements executed if the expression equals the
        // value of this constant_expression
        break;
    default:
        // statements executed if expression does not equal
        // any case constant_expression
}
```

You can use the `break` statement to end processing of a particular labeled statement within the `switch` statement. It branches to the end of the `switch` statement. Without `break`, the program continues to the next labeled statement, executing the statements until a `break` or the end of the statement is reached. This continuation may be desirable in some situations.

The `default` statement is executed if no `case` *`constant-expression`* value is equal to the value of *`expression`*. If there's no `default` statement, and no `case` match is found, none of the statements in the `switch` body get executed. There can be at most one `default` statement. The `default` statement doesn't have to come at the end. It may appear anywhere in the body of the `switch` statement. A `case` or `default` label can only appear inside a `switch` statement.

The type of `switch` *`expression`* and `case` *`constant-expression`* must be integral. The value of each `case` *`constant-expression`* must be unique within the statement body.

The `case` and `default` labels of the `switch` statement's body are significant only in the initial test that determines where execution starts in the statement body. `switch` statements can be nested. Any static variables are initialized before executing into any `switch` statements.

> **NOTE**
>
> Declarations can appear at the head of the compound statement forming the `switch` body, but initializations included in the declarations are not performed. The `switch` statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

The following examples illustrate `switch` statements:

```
switch( c )
{
    case 'A':
        capital_a++;
    case 'a':
        letter_a++;
    default :
        total++;
}
```

All three statements of the `switch` body in this example are executed if `c` is equal to `'A'`, since no `break` statement appears before the following `case`. Execution control is transferred to the first statement (`capital_a++;`) and continues in order through the rest of the body. If `c` is equal to `'a'`, `letter_a` and `total` are incremented. Only `total` is incremented when `c` doesn't equal `'A'` or `'a'`.

```
switch( i )
{
    case -1:
        n++;
        break;
    case 0 :
        z++;
        break;
    case 1 :
        p++;
        break;
}
```

In this example, a `break` statement follows each statement of the `switch` body. The `break` statement forces an exit from the statement body after one statement is executed. If `i` is equal to -1, only `n` is incremented. The `break` following the statement `n++;` causes execution control to pass out of the statement body, bypassing the remaining statements. Similarly, if `i` is equal to 0, only `z` is incremented; if `i` is equal to 1, only `p` is incremented. The final `break` statement isn't strictly necessary, since control passes out of the body at the end of the compound statement. It's included for consistency.

A single statement can carry multiple `case` labels, as the following example shows:

```
switch( c )
{
    case 'a' :
    case 'b' :
    case 'c' :
    case 'd' :
    case 'e' :
    case 'f' :  convert_hex(c);
}
```

In this example, if *constant-expression* equals any letter between `'a'` and `'f'`, the `convert_hex` function is called.

Microsoft-specific

Microsoft C doesn't limit the number of `case` values in a `switch` statement. The number is limited only by the available memory. ANSI C requires at least 257 `case` labels be allowed in a `switch` statement.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the /Za compiler option to disable these extensions.

See also

switch Statement (C++)

try-except statement (C)

12/22/2021 • 3 minutes to read • Edit Online

Microsoft-specific

The `try-except` statement is a Microsoft extension to the C language that enables applications to gain control of a program when events that normally terminate execution occur. Such events are called exceptions, and the mechanism that deals with exceptions is called structured exception handling.

Exceptions may be either hardware- or software-based. Even when applications can't completely recover from hardware or software exceptions, structured exception handling makes it possible to log and display error information. It's useful to trap the internal state of the application to help diagnose the problem. In particular, it's helpful for intermittent problems that aren't easy to reproduce.

Syntax

> `try-except-statement` :
>   `__try` `compound-statement` `__except (` `expression` `)` `compound-statement`

The compound statement after the `__try` clause is the *guarded section*. The compound statement after the `__except` clause is the *exception handler*. The handler specifies a set of actions to take if an exception is raised during execution of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the `__except` clause.
3. If an exception occurs during execution of the guarded section, or in any routine the guarded section calls, the `__except` expression gets evaluated. The value returned determines how the exception is handled. There are three possible values:
   - `EXCEPTION_CONTINUE_SEARCH` : The exception isn't recognized. Continue to search up the stack for a handler, first for containing `try-except` statements, then for handlers with the next highest precedence.
   - `EXCEPTION_CONTINUE_EXECUTION` : The exception is recognized but dismissed. Continue execution at the point where the exception occurred.
   - `EXCEPTION_EXECUTE_HANDLER` The exception is recognized. Transfer control to the exception handler by executing the `__except` compound statement, then continue execution at the point the exception occurred.

Because the `__except` expression is evaluated as a C expression, it's limited to either a single value, the conditional-expression operator, or the comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

> **NOTE**
>
> Structured exception handling works with C and C++ source files. However, it isn't specifically designed for C++. For portable C++ programs, C++ exception handling should be used instead of structured exception handling. Also, the C++ exception handling mechanism is much more flexible, in that it can handle exceptions of any type. For more information, see Exception handling in the *C++ Language Reference*.

Each routine in an application can have its own exception handler. The `__except` expression executes in the scope of the `__try` body. It has access to any local variables declared there.

The `__leave` keyword is valid within a `try-except` statement block. The effect of `__leave` is to jump to the end of the `try-except` block. Execution resumes after the end of the exception handler. Although a `goto` statement can be used to accomplish the same result, a `goto` statement causes stack unwinding. The `__leave` statement is more efficient because it doesn't involve stack unwinding.

Exiting a `try-except` statement using the `longjmp` run-time function is considered abnormal termination. It isn't legal to jump into a `__try` statement, but it's legal to jump out of one. The exception handler isn't called if a process is killed in the middle of executing a `try-except` statement.

Example

Here's an example of an exception handler and a termination handler. For more information about termination handlers, see `try-finally` statement (C).

```
.
.
.
puts("hello");
__try {
   puts("in try");
   __try {
      puts("in try");
      RAISE_AN_EXCEPTION();
   } __finally {
      puts("in finally");
   }
} __except( puts("in filter"), EXCEPTION_EXECUTE_HANDLER ) {
   puts("in except");
}
puts("world");
```

Here's the output from the example, with commentary added on the right:

```
hello
in try              /* fall into try                     */
in try              /* fall into nested try              */
in filter           /* execute filter; returns 1 so accept  */
in finally          /* unwind nested finally             */
in except           /* transfer control to selected handler */
world               /* flow out of handler               */
```

END Microsoft-specific

See also

`try-except` statement (C++)

try-finally statement (C)

12/22/2021 • 2 minutes to read • Edit Online

Microsoft-specific

The `try-finally` statement is a Microsoft extension to the C language that enables applications to guarantee execution of cleanup code when execution of a block of code is interrupted. Cleanup consists of such tasks as deallocating memory, closing files, and releasing file handles. The `try-finally` statement is especially useful for routines that have several places where a check is made for an error that could cause premature return from the routine.

> *`try-finally-statement`* :
> `__try` *`compound-statement`* `__finally` *`compound-statement`*

The compound statement after the `__try` clause is the guarded section. The compound statement after the `__finally` clause is the termination handler. The handler specifies a set of actions that execute when the guarded section is exited. It doesn't matter whether the guarded section is exited by an exception (abnormal termination) or by standard fall through (normal termination).

Control reaches a `__try` statement by simple sequential execution (fall through). When control enters the `__try` statement, its associated handler becomes active. Execution proceeds as follows:

1. The guarded section is executed.
2. The termination handler is invoked.
3. When the termination handler completes, execution continues after the `__finally` statement. No matter how the guarded section ends (for example, via a `goto` statement out of the guarded body or via a `return` statement), the termination handler is executed before the flow of control moves out of the guarded section.

The `__leave` keyword is valid within a `try-finally` statement block. The effect of `__leave` is to jump to the end of the `try-finally` block. The termination handler is immediately executed. Although a `goto` statement can be used to accomplish the same result, a `goto` statement causes stack unwinding. The `__leave` statement is more efficient because it doesn't involve stack unwinding.

Exiting a `try-finally` statement using a `return` statement or the `longjmp` run-time function is considered abnormal termination. It's not legal to jump into a `__try` statement, but legal to jump out of one. All `__finally` statements that are active between the point of departure and the destination must be run. It's called a *local unwind*.

The termination handler isn't called if a process is killed while executing a `try-finally` statement.

> **NOTE**
>
> Structured exception handling works with C and C++ source files. However, it isn't specifically designed for C++. For portable C++ programs, C++ exception handling should be used instead of structured exception handling. Also, the C++ exception handling mechanism is much more flexible, in that it can handle exceptions of any type. For more information, see Exception handling in the *C++ Language Reference*.

See the example for the `try-except` statement to see how the `try-finally` statement works.

END Microsoft-specific

See also

`try-finally` statement (C++)

while Statement (C)

12/22/2021 • 2 minutes to read • Edit Online

The `while` statement lets you repeat a statement until a specified expression becomes false.

Syntax

iteration-statement:
  while (*expression* **)** *statement*

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The *expression* is evaluated.
2. If *expression* is initially false, the body of the `while` statement is never executed, and control passes from the `while` statement to the next statement in the program.

   If *expression* is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.

The `while` statement can also terminate when a `break`, `goto`, or `return` within the statement body is executed. Use the `continue` statement to terminate an iteration without exiting the `while` loop. The `continue` statement passes control to the next iteration of the `while` statement.

This is an example of the `while` statement:

```
while ( i >= 0 )
{
    string1[i] = string2[i];
    i--;
}
```

This example copies characters from `string2` to `string1`. If `i` is greater than or equal to 0, `string2[i]` is assigned to `string1[i]` and `i` is decremented. When `i` reaches or falls below 0, execution of the `while` statement terminates.

See also

while Statement (C++)

Functions (C)

12/22/2021 • 2 minutes to read • Edit Online

The function is the fundamental modular unit in C. A function is usually designed to perform a specific task, and its name often reflects that task. A function contains declarations and statements. This section describes how to declare, define, and call C functions. Other topics discussed are:

- Overview of functions
- Function attributes
- Specifying calling conventions
- Inline functions
- DLL export and import functions
- Naked functions
- Storage class
- Return type
- Arguments
- Parameters

See also

C Language Reference

Overview of Functions

12/22/2021 • 2 minutes to read • Edit Online

Functions must have a definition and should have a declaration, although a definition can serve as a declaration if the declaration appears before the function is called. The function definition includes the function body — the code that executes when the function is called.

A function declaration establishes the name, return type, and attributes of a function that is defined elsewhere in the program. A function declaration must precede the call to the function. This is why the header files containing the declarations for the run-time functions are included in your code before a call to a run-time function. If the declaration has information about the types and number of parameters, the declaration is a prototype. See Function Prototypes for more information.

The compiler uses the prototype to compare the types of arguments in subsequent calls to the function with the function's parameters and to convert the types of the arguments to the types of the parameters whenever necessary.

A function call passes execution control from the calling function to the called function. The arguments, if any, are passed by value to the called function. Execution of a `return` statement in the called function returns control and possibly a value to the calling function.

See also

Functions

Obsolete Forms of Function Declarations and Definitions

12/22/2021 • 2 minutes to read • Edit Online

The old-style function declarations and definitions use slightly different rules for declaring parameters than the syntax recommended by the ANSI C standard. First, the old-style declarations don't have a parameter list. Second, in the function definition, the parameters are listed, but their types are not declared in the parameter list. The type declarations precede the compound statement constituting the function body. The old-style syntax is obsolete and should not be used in new code. Code using the old-style syntax is still supported, however. This example illustrates the obsolete forms of declarations and definitions:

```
double old_style();           /* Obsolete function declaration */

double alt_style( a , real )  /* Obsolete function definition */
    double *real;
    int a;
{
    return ( *real + a ) ;
}
```

Functions returning an integer or pointer with the same size as an `int` are not required to have a declaration although the declaration is recommended.

To conform to the ANSI C standard, old-style function declarations using an ellipsis now generate an error when compiling with the /Za option and a level 4 warning when compiling with /Ze. For example:

```
void funct1( a, ... )  /* Generates a warning under /Ze or */
int a;                 /* an error when compiling with /Za */
{
}
```

You should rewrite this declaration as a prototype:

```
void funct1( int a, ... )
{
}
```

Old-style function declarations also generate warnings if you subsequently declare or define the same function with either an ellipsis or a parameter with a type that is not the same as its promoted type.

The next section, C Function Definitions, shows the syntax for function definitions, including the old-style syntax. The nonterminal for the list of parameters in the old-style syntax is *identifier-list*.

See also

Overview of Functions

C Function Definitions

12/22/2021 • 2 minutes to read • Edit Online

A function definition specifies the name of the function, the types and number of parameters it expects to receive, and its return type. A function definition also includes a function body with the declarations of its local variables, and the statements that determine what the function does.

Syntax

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration: /* Allowed only at external (file) scope */
  function-definition
  declaration

function-definition:
  declaration-specifiers$_{\text{opt}}$ *attribute-seq*$_{\text{opt}}$ *declarator declaration-list*$_{\text{opt}}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

Prototype parameters are:

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{\text{opt}}$
  type-specifier declaration-specifiers$_{\text{opt}}$
  type-qualifier declaration-specifiers$_{\text{opt}}$

declaration-list:
  declaration
  declaration-list declaration

declarator:
  pointer$_{\text{opt}}$ *direct-declarator*

direct-declarator: /* A function declarator */
  direct-declarator **(** *parameter-type-list* **)** /* New-style declarator */
  direct-declarator **(** *identifier-list*$_{\text{opt}}$ **)** /* Obsolete-style declarator */

The parameter list in a definition uses this syntax:

parameter-type-list: /* The parameter list */
  parameter-list
  parameter-list **, ...**

parameter-list:
  parameter-declaration
  parameter-list **,** *parameter-declaration*

parameter-declaration:
  declaration-specifiers declarator
  declaration-specifiers abstract-declarator$_{\text{opt}}$

The parameter list in an old-style function definition uses this syntax:

identifier-list: /* Used in obsolete-style function definitions and declarations */
  identifier
  identifier-list , *identifier*

The syntax for the function body is:

compound-statement:
  { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

The only storage-class specifiers that can modify a function declaration are `extern` and `static`. The `extern` specifier signifies that the function can be referenced from other files; that is, the function name is exported to the linker. The `static` specifier signifies that the function cannot be referenced from other files; that is, the name is not exported by the linker. If no storage class appears in a function definition, `extern` is assumed. In any case, the function is always visible from the definition point to the end of the file.

The optional *declaration-specifiers* and mandatory *declarator* together specify the function's return type and name. The *declarator* is a combination of the identifier that names the function and the parentheses following the function name. The optional *attribute-seq* nonterminal is a Microsoft-specific feature defined in Function Attributes.

The *direct-declarator* (in the *declarator* syntax) specifies the name of the function being defined and the identifiers of its parameters. If the *direct-declarator* includes a *parameter-type-list*, the list specifies the types of all the parameters. Such a declarator also serves as a function prototype for later calls to the function.

A *declaration* in the *declaration-list* in function definitions cannot contain a *storage-class-specifier* other than `register`. The *type-specifier* in the *declaration-specifiers* syntax can be omitted only if the `register` storage class is specified for a value of `int` type.

The *compound-statement* is the function body containing local variable declarations, references to externally declared items, and statements.

The sections Function Attributes, Storage Class, Return Type, Parameters, and Function Body describe the components of the function definition in detail.

See also

Functions

Function Attributes

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The optional *attribute-seq* nonterminal allows you to select a calling convention on a per-function basis. You can also specify functions as `__fastcall` or `__inline`.

END Microsoft Specific

See also

C Function Definitions

Specifying Calling Conventions

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

For information on calling conventions, see Calling Conventions Topics.

END Microsoft Specific

See also

Function Attributes

Inline Functions

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The `__inline` keyword tells the compiler to substitute the code within the function definition for every instance of a function call. However, substitution occurs only at the compiler's discretion. For example, the compiler does not inline a function if its address is taken or if it is too large to inline.

For a function to be considered as a candidate for inlining, it must use the new-style function definition.

Use this form to specify an inline function:

> `__inline` *type*$_{\text{opt}}$ *function-definition*

The use of inline functions generates faster code and can sometimes generate smaller code than the equivalent function call generates for the following reasons:

- It saves the time required to execute function calls.
- Small inline functions, perhaps three lines or less, create less code than the equivalent function call because the compiler doesn't generate code to handle arguments and a return value.
- Functions generated inline are subject to code optimizations not available to normal functions because the compiler does not perform interprocedural optimizations.

Functions using `__inline` should not be confused with inline assembler code. See Inline Assembler for more information.

END Microsoft Specific

See also

inline, __inline, __forceinline

Inline Assembler (C)

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The inline assembler lets you embed assembly-language instructions directly in your C source programs without extra assembly and link steps. The inline assembler is built into the compiler — you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C variable or function name that is in scope, so it is easy to integrate it with your program's C code. And because the assembly code can be mixed with C statements, it can do tasks that are cumbersome or impossible in C alone.

The `__asm` keyword invokes the inline assembler and can appear wherever a C statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "`__asm` block" here refers to any instruction or group of instructions, whether or not in braces.

The code below is a simple `__asm` block enclosed in braces. (The code is a custom function prolog sequence.)

```
__asm
{
   push ebp
   mov  ebp, esp
   sub  esp, __LOCAL_SIZE
}
```

Alternatively, you can put `__asm` in front of each assembly instruction:

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

Since the `__asm` keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

END Microsoft Specific

See also

Function Attributes

`_Noreturn` keyword and `noreturn` macro (C11)

12/22/2021 • 2 minutes to read • Edit Online

The `_Noreturn` keyword was introduced in C11. It tells the compiler that the function it's applied to doesn't return to the caller. The compiler knows that the code following a call to a `_Noreturn` function is unreachable. An example of a function that doesn't return is abort. If there's a possibility for control flow to return to the caller, the function must not have the `_Noreturn` attribute.

The keyword is typically used through the convenience macro, `noreturn`, provided in <stdnoreturn.h>, which maps to the `_Noreturn` keyword.

The primary benefits for using `_Noreturn` (or the equivalent `noreturn`) are making the intention of the function clear in the code for future readers, and detecting unintentionally unreachable code.

A function marked `noreturn` shouldn't include a return type because it doesn't return a value to the caller. It should be `void`.

Example using `noreturn` macro and `_Noreturn` keyword

The following example demonstrates the `_Noreturn` keyword and the equivalent `noreturn` macro.

IntelliSense may generate a spurious error, `E0065`, if you use the macro `noreturn` that you can ignore. It doesn't prevent you from running the sample.

```
// Compile with Warning Level4 (/W4) and /std:c11
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void fatal_error(void)
{
    exit(3);
}

_Noreturn void not_coming_back(void)
{
    puts("There's no coming back");
    fatal_error();
    return; // warning C4645 - function declared with noreturn has a return statement
}

void done(void)
{
    puts("We'll never get here");
}

int main(void)
{
    not_coming_back();
    done(); // warning c4702 - unreachable code

    return 0;
}
```

Requirements

MACRO	REQUIRED HEADER
`noreturn`	<stdnoreturn.h>

See also

/std (Specify language standard version)
/W4 (Specify warning level)
C4702 warning
__declspec(noreturn)

DLL Import and Export Functions

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The most complete and up-to-date information on this topic can be found in dllexport, dllimport.

The `dllimport` and `dllexport` storage-class modifiers are Microsoft-specific extensions to the C language. These modifiers explicitly define the DLL's interface to its client (the executable file or another DLL). Declaring functions as `dllexport` eliminates the need for a module-definition (.DEF) file. You can also use the `dllimport` and `dllexport` modifiers with data and objects.

The `dllimport` and `dllexport` storage-class modifiers must be used with the extended attribute syntax keyword, `__declspec`, as shown in this example:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllExport int j;
DllExport int n;
```

For specific information about the syntax for extended storage-class modifiers, see Extended Storage-Class Attributes.

END Microsoft Specific

See also

C Function Definitions

Definitions and Declarations (C)

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The DLL interface refers to all items (functions and data) that are known to be exported by some program in the system; that is, all items that are declared as `dllimport` or `dllexport`. All declarations included in the DLL interface must specify either the `dllimport` or `dllexport` attribute. However, the definition can specify only the `dllexport` attribute. For example, the following function definition generates a compiler error:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int func()    /* Error; dllimport prohibited in */
                        /* definition. */
{
   return 1;
}
```

This code also generates an error:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int i = 10;      /* Error; this is a definition. */
```

However, this is correct syntax:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport int i = 10;      /* Okay: this is an export definition. */
```

The use of `dllexport` implies a definition, while `dllimport` implies a declaration. You must use the `extern` keyword with `dllexport` to force a declaration; otherwise, a definition is implied.

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllImport int k;   /* These are correct and imply */
Dllimport int j;          /* a declaration. */
```

END Microsoft Specific

See also

DLL Import and Export Functions

Defining Inline C Functions with dllexport and dllimport

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

You can define as inline a function with the `dllexport` attribute. In this case, the function is always instantiated and exported, whether or not any module in the program references the function. The function is presumed to be imported by another program.

You can also define as inline a function declared with the `dllimport` attribute. In this case, the function can be expanded (subject to the /Ob (inline) compiler option specification) but never instantiated. In particular, if the address of an inline imported function is taken, the address of the function residing in the DLL is returned. This behavior is the same as taking the address of a non-inline imported function.

Static local data and strings in inline functions maintain the same identities between the DLL and client as they would in a single program (that is, an executable file without a DLL interface).

Exercise care when providing imported inline functions. For example, if you update the DLL, don't assume that the client will use the changed version of the DLL. To ensure that you are loading the proper version of the DLL, rebuild the DLL's client as well.

END Microsoft Specific

See also

DLL Import and Export Functions

Rules and Limitations for dllimport/dllexport

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

- If you declare a function without the `dllimport` or `dllexport` attribute, the function is not considered part of the DLL interface. Therefore, the definition of the function must be present in that module or in another module of the same program. To make the function part of the DLL interface, you must declare the definition of the function in the other module as `dllexport`. Otherwise, a linker error is generated when the client is built.
- If a single module in your program contains `dllimport` and `dllexport` declarations for the same function, the `dllexport` attribute takes precedence over the `dllimport` attribute. However, a compiler warning is generated. For example:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllExport void func1( void );   /* Warning; dllexport */
                                   /* takes precedence. */
```

- You cannot initialize a static data pointer with the address of a data object declared with the `dllimport` attribute. For example, the following code generates errors:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport int i;
   .
   .
   .
   int *pi = &i;                           /* Error */

   void func2()
   {
      static int *pi = &i;                  /* Error */
   }
```

- Initializing a static function pointer with the address of a function declared with `dllimport` sets the pointer to the address of the DLL import thunk (a code stub that transfers control to the function) rather than the address of the function. This assignment does not generate an error message:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void
   .
   .
   .
   static void ( *pf )( void ) = &func1;   /* No Error */

   void func2()
   {
      static void ( *pf )( void ) = &func1;  /* No Error */
   }
```

- Because a program that includes the `dllexport` attribute in the declaration of an object must provide the definition for that object, you can initialize a global or local static function pointer with the address of a `dllexport` function. Similarly, you can initialize a global or local static data pointer with the address of a `dllexport` data object. For example:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllImport int i;

   DllExport void func1( void );
   DllExport int i;
   .
   .
   .
   int *pi = &i;                            /* Okay */
   static void ( *pf )( void ) = &func1;    /* Okay */

   void func2()
   {
      static int *pi = i;                     /* Okay */
      static void ( *pf )( void ) = &func1;   /* Okay */
   }
```

END Microsoft Specific

See also

DLL Import and Export Functions

Naked Functions

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

The `naked` storage-class attribute is a Microsoft-specific extension to the C language. For functions declared with the `naked` storage-class attribute, the compiler generates code without prolog and epilog code. You can use this feature to write your own prolog/epilog code sequences using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

Because the `naked` attribute is only relevant to the definition of a function and is not a type modifier, naked functions use the extended attribute syntax, described in Extended Storage-Class Attributes.

The following example defines a function with the `naked` attribute:

```
__declspec( naked ) int func( formal_parameters )
{
   /* Function body */
}
```

Or, alternatively:

```
#define Naked   __declspec( naked )

Naked int func( formal_parameters )
{
   /* Function body */
}
```

The `naked` attribute affects only the nature of the compiler's code generation for the function's prolog and epilog sequences. It does not affect the code that is generated for calling such functions. Thus, the `naked` attribute is not considered part of the function's type, and function pointers cannot have the `naked` attribute. Furthermore, the `naked` attribute cannot be applied to a data definition. For example, the following code generates errors:

```
__declspec( naked ) int i;  /* Error--naked attribute not */
                            /* permitted on data declarations. */
```

The `naked` attribute is relevant only to the definition of the function and cannot be specified in the function's prototype. The following declaration generates a compiler error:

```
__declspec( naked ) int func();   /* Error--naked attribute not */
                     /* permitted on function declarations.    */    \
```

END Microsoft Specific

See also

C Function Definitions

Rules and Limitations for Using Naked Functions

12/22/2021 • 2 minutes to read • Edit Online

For information on rules and limitations for using naked functions, see the corresponding topic in the C++ language reference: Rules and Limitations for Naked Functions.

See also

Naked Functions

Considerations When Writing Prolog/Epilog Code

12/22/2021 • 2 minutes to read • Edit Online

Microsoft Specific

Before writing your own prolog and epilog code sequences, it is important to understand how the stack frame is laid out. It is also useful to know how to use the **__LOCAL_SIZE** predefined constant.

CStack Frame Layout

This example shows the standard prolog code that might appear in a 32-bit function:

```
push     ebp                    ; Save ebp
mov      ebp, esp               ; Set stack frame pointer
sub      esp, localbytes        ; Allocate space for locals
push     <registers>            ; Save registers
```

The `localbytes` variable represents the number of bytes needed on the stack for local variables, and the `registers` variable is a placeholder that represents the list of registers to be saved on the stack. After pushing the registers, you can place any other appropriate data on the stack. The following is the corresponding epilog code:

```
pop      <registers>            ; Restore registers
mov      esp, ebp               ; Restore stack pointer
pop      ebp                    ; Restore ebp
ret                             ; Return from function
```

The stack always grows down (from high to low memory addresses). The base pointer (`ebp`) points to the pushed value of `ebp` . The local variables area begins at `ebp-2` . To access local variables, calculate an offset from `ebp` by subtracting the appropriate value from `ebp` .

The __LOCAL_SIZE Constant

The compiler provides a constant, **__LOCAL_SIZE**, for use in the inline assembler block of function prolog code. This constant is used to allocate space for local variables on the stack frame in custom prolog code.

The compiler determines the value of **__LOCAL_SIZE**. The value is the total number of bytes of all user-defined local variables and compiler-generated temporary variables. **__LOCAL_SIZE** can be used only as an immediate operand; it cannot be used in an expression. You must not change or redefine the value of this constant. For example:

```
mov      eax, __LOCAL_SIZE           ;Immediate operand--Okay
mov      eax, [ebp - __LOCAL_SIZE]   ;Error
```

The following example of a naked function containing custom prolog and epilog sequences uses **__LOCAL_SIZE** in the prolog sequence:

```
__declspec ( naked ) func()
{
   int i;
   int j;

   __asm      /* prolog */
      {
      push   ebp
      mov      ebp, esp
      sub      esp, __LOCAL_SIZE
      }

   /* Function body */

   __asm      /* epilog */
      {
      mov      esp, ebp
      pop      ebp
      ret
      }
}
```

END Microsoft Specific

See also

Naked Functions

Storage Class

12/22/2021 • 2 minutes to read • Edit Online

The storage-class specifier in a function definition gives the function either `extern` or `static` storage class.

Syntax

function-definition:
  declaration-specifiers$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

storage-class-specifier: /* For function definitions */
  `extern`
  `static`

If a function definition does not include a *storage-class-specifier*, the storage class defaults to `extern`. You can explicitly declare a function as `extern`, but it is not required.

If the declaration of a function 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 visible declaration with file scope, the identifier has external linkage. If an identifier has file scope and no *storage-class-specifier*, the identifier has external linkage. External linkage means that each instance of the identifier denotes the same object or function. See Lifetime, Scope, Visibility, and Linkage for more information about linkage and file scope.

Block-scope function declarations with a storage-class specifier other than `extern` generate errors.

A function with `static` storage class is visible only in the source file in which it is defined. All other functions, whether they are given `extern` storage class explicitly or implicitly, are visible throughout all source files in the program. If `static` storage class is desired, it must be declared on the first occurrence of a declaration (if any) of the function, and on the definition of the function.

Microsoft Specific

When the Microsoft extensions are enabled, a function originally declared without a storage class (or with `extern` storage class) is given `static` storage class if the function definition is in the same source file and if the definition explicitly specifies `static` storage class.

When compiling with the /Ze compiler option, functions declared within a block using the `extern` keyword have global visibility. This is not true when compiling with /Za. This feature should not be relied upon if portability of source code is a consideration.

END Microsoft Specific

See also

C Function Definitions

Return Type

12/22/2021 • 2 minutes to read • Edit Online

The return type of a function establishes the size and type of the value returned by the function and corresponds to the type-specifier in the syntax below:

Syntax

function-definition:
  declaration-specifiers$_{\text{opt}}$ *attribute-seq*$_{\text{opt}}$ *declarator* *declaration-list*$_{\text{opt}}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

declaration-specifiers:
  storage-class-specifier *declaration-specifiers*$_{\text{opt}}$
  type-specifier *declaration-specifiers*$_{\text{opt}}$
  type-qualifier *declaration-specifiers*$_{\text{opt}}$

type-specifier:
  `void`
  `char`
  `short`
  `int`
  `__int8` /* Microsoft-specific */
  `__int16` /* Microsoft-specific */
  `__int32` /* Microsoft-specific */
  `__int64` /* Microsoft-specific */
  `long`
  `float`
  `double`
  `signed`
  `unsigned`
  struct-or-union-specifier
  enum-specifier
  typedef-name

The *type-specifier* can specify any fundamental, structure, or union type. If you do not include *type-specifier*, the return type `int` is assumed.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. A function returns a value when a `return` statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point at which the function was called. If a function is declared with return type `void`, a return statement containing an expression generates a warning and the expression is not evaluated.

The following examples illustrate function return values.

```c
typedef struct
{
    char name[20];
    int id;
    long class;
} STUDENT;

/* Return type is STUDENT: */

STUDENT sortstu( STUDENT a, STUDENT b )
{
    return ( (a.id < b.id) ? a : b );
}
```

This example defines the `STUDENT` type with a `typedef` declaration and defines the function `sortstu` to have `STUDENT` return type. The function selects and returns one of its two structure arguments. In subsequent calls to the function, the compiler checks to make sure the argument types are `STUDENT`.

NOTE

Efficiency would be enhanced by passing pointers to the structure, rather than the entire structure.

```c
char *smallstr( char s1[], char s2[] )
{
    int i;

    i = 0;
    while ( s1[i] != '\0' && s2[i] != '\0' )
        i++;
    if ( s1[i] == '\0' )
        return ( s1 );
    else
        return ( s2 );
}
```

This example defines a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the first of the array elements and has its type; thus, the return type of the function is a pointer to type `char`.

You need not declare functions with `int` return type before you call them, although prototypes are recommended so that correct type checking for arguments and return values is enabled.

See also

C Function Definitions

Parameters

12/22/2021 • 2 minutes to read • Edit Online

Arguments are names of values passed to a function by a function call. Parameters are the values the function expects to receive. In a function prototype, the parentheses following the function name contain a complete list of the function's parameters and their types. Parameter declarations specify the types, sizes, and identifiers of values stored in the parameters.

Syntax

`function-definition` :
  `declaration-specifiers` opt `attribute-seq` opt `declarator` `declaration-list` opt `compound-statement`

/* `attribute-seq` is Microsoft-specific */

`declarator` :
  `pointer` opt `direct-declarator`

`direct-declarator` : /* A function declarator */
  `direct-declarator` `(` `parameter-type-list` `)` /* New-style declarator */
  `direct-declarator` `(` `identifier-list` opt `)` /* Obsolete-style declarator */

`parameter-type-list` : /* The parameter list */
  `parameter-list`
  `parameter-list` `, ...`

`parameter-list` :
  `parameter-declaration`
  `parameter-list` `,` `parameter-declaration`

`parameter-declaration` :
  `declaration-specifiers` `declarator`
  `declaration-specifiers` `abstract-declarator` opt

The `parameter-type-list` is a sequence of parameter declarations separated by commas. The form of each parameter in a parameter list looks like this:

> `register` opt `type-specifier` `declarator` opt

Function parameters declared with the `auto` attribute generate errors. The identifiers of the parameters are used in the function body to refer to the values passed to the function. You can name the parameters in a prototype, but the names go out of scope at the end of the declaration. That means parameter names can be assigned the same way or differently in the function definition. These identifiers can't be redefined in the outermost block of the function body, but they can be redefined in inner, nested blocks as though the parameter list were an enclosing block.

Each identifier in `parameter-type-list` must be preceded by its appropriate type specifier, as shown in this example:

```
void new( double x, double y, double z )
{
    /* Function body here */
}
```

If at least one parameter occurs in the parameter list, the list can end with a comma followed by three periods (`, ...`). This construction, called the "ellipsis notation," indicates a variable number of arguments to the function. (For more information, see Calls with a Variable Number of Arguments.) However, a call to the function must have at least as many arguments as there are parameters before the last comma.

If no arguments are to be passed to the function, the list of parameters is replaced by the keyword `void`. This use of `void` is distinct from its use as a type specifier.

The order and type of parameters, including any use of the ellipsis notation, must be the same in all the function declarations (if any) and in the function definition. The types of the arguments after usual arithmetic conversions must be assignment-compatible with the types of the corresponding parameters. (See Usual Arithmetic Conversions for information on arithmetic conversions.) Arguments following the ellipsis aren't checked. A parameter can have any fundamental, structure, union, pointer, or array type.

The compiler performs the usual arithmetic conversions independently on each parameter and on each argument, if necessary. After conversion, no parameter is shorter than an `int`, and no parameter has `float` type unless the parameter type is explicitly specified as `float` in the prototype. It means, for example, that declaring a parameter as a `char` has the same effect as declaring it as an `int`.

See also

C Function Definitions

Function Body

12/22/2021 • 2 minutes to read • Edit Online

A *function body* is a compound statement containing the statements that specify what the function does.

Syntax

function-definition:
  declaration-specifiers$_{opt}$ *attribute-seq*$_{opt}$ *declarator* *declaration-list*$_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

compound-statement: /* The function body */
  { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

Variables declared in a function body, known as *local variables*, have `auto` storage class unless otherwise specified. When the function is called, storage is created for the local variables and local initializations are performed. Execution control passes to the first statement in *compound-statement* and continues until a `return` statement is executed or the end of the function body is encountered. Control then returns to the point at which the function was called.

A `return` statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no `return` statement is executed or if the `return` statement does not include an expression.

See also

C Function Definitions

Function Prototypes

12/22/2021 • 3 minutes to read • Edit Online

A function declaration precedes the function definition and specifies the name, return type, storage class, and other attributes of a function. To be a prototype, the function declaration must also establish types and identifiers for the function's arguments.

Syntax

declaration:
  declaration-specifiers attribute-seq$_{opt}$ *init-declarator-list*$_{opt}$ **;**

/* *attribute-seq*$_{opt}$ is Microsoft-specific */

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator:
  declarator
  declarator **=** *initializer*

declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator: /* A function declarator */
  direct-declarator **(** *parameter-type-list* **)** /* New-style declarator */
  direct-declarator **(** *identifier-list*$_{opt}$ **)** /* Obsolete-style declarator */

The prototype has the same form as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body. In either case, the return type must agree with the return type specified in the function definition.

Function prototypes have the following important uses:

- They establish the return type for functions that return types other than `int`. Although functions that return `int` values don't require prototypes, prototypes are recommended.
- Without complete prototypes, standard conversions are made, but no attempt is made to check the type or number of arguments with the number of parameters.
- Prototypes are used to initialize pointers to functions before those functions are defined.
- The parameter list is used to check that arguments in the function call match the parameters in the function definition.

The converted type of each parameter determines the interpretation of the arguments that the function call places on the stack. A type mismatch between an argument and a parameter may cause the arguments on the stack to be misinterpreted. For example, on a 16-bit computer, if a 16-bit pointer is passed as an argument, then declared as a `long` parameter, the first 32 bits on the stack are interpreted as a `long` parameter. This error creates problems not only with the `long` parameter, but with any parameters that follow it. You can detect errors of this kind by declaring complete function prototypes for all functions.

A prototype establishes the attributes of a function. Then, function calls that precede the function definition (or that occur in other source files) can be checked for argument-type and return-type mismatches. For example, if you specify the `static` storage-class specifier in a prototype, you must also specify the `static` storage class in the function definition.

Complete parameter declarations (`int a`) can be mixed with abstract declarators (`int`) in the same declaration. For example, the following declaration is legal:

```
int add( int a, int );
```

The prototype can include both the type of, and an identifier for, each expression that's passed as an argument. However, such identifiers are only in scope until the end of the declaration. The prototype can also reflect the fact that the number of arguments is variable, or that no arguments are passed. Without such a list, mismatches may not be revealed, so the compiler can't generate diagnostic messages concerning them. For more information on type checking, see Arguments.

Prototype scope in the Microsoft C compiler is now ANSI-conforming when compiling with the `/Za` compiler option. If you declare a `struct` or `union` tag within a prototype, the tag is entered at that scope rather than at global scope. For example, when compiling with `/Za` for ANSI conformance, you can never call this function without getting a type mismatch error:

```
void func1( struct S * );
```

To correct your code, define or declare the `struct` or `union` at global scope before the function prototype:

```
struct S;
void func1( struct S * );
```

Under **/Ze**, the tag is still entered at global scope.

See also

Functions

Function Calls

12/22/2021 • 2 minutes to read • Edit Online

A *function call* is an expression that passes control and arguments (if any) to a function and has the form:

expression (*expression-list*$_{opt}$)

where *expression* is a function name or evaluates to a function address and *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the arguments passed to the function. If the function does not return a value, then you declare it to be a function that returns `void`.

If a declaration exists before the function call, but no information is given concerning the parameters, any undeclared arguments simply undergo the usual arithmetic conversions.

> **NOTE**
>
> The expressions in the function argument list can be evaluated in any order, so arguments whose values may be changed by side effects from another argument have undefined values. The sequence point defined by the function-call operator guarantees only that all side effects in the argument list are evaluated before control passes to the called function. (Note that the order in which arguments are pushed on the stack is a separate matter.) See Sequence Points for more information.

The only requirement in any function call is that the expression before the parentheses must evaluate to a function address. This means that a function can be called through any function-pointer expression.

Example

This example illustrates function calls called from a `switch` statement:

```
int main()
{
    /* Function prototypes */

    long lift( int ), step( int ), drop( int );
    void work( int number, long (*function)(int i) );

    int select, count;
    .
    .
    .
    select = 1;
    switch( select )
    {
        case 1: work( count, lift );
                break;

        case 2: work( count, step );
                break;

        case 3: work( count, drop );
                /* Fall through to next case */
        default:
                break;
    }
}

/* Function definition */

void work( int number, long (*function)(int i) )
{
    int i;
    long j;

    for ( i = j = 0; i < number; i++ )
            j += ( *function )( i );
}
```

In this example, the function call in `main`,

```
work( count, lift );
```

passes an integer variable, `count`, and the address of the function `lift` to the function `work`. Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a prototype for `work` is given at the beginning of the `main` function.

The parameter `function` in `work` is declared to be a pointer to a function taking one `int` argument and returning a `long` value. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a `long` value.

The function `work` calls the selected function from inside the `for` loop by using the following function call:

```
( *function )( i );
```

One argument, `i`, is passed to the called function.

See also

Arguments

12/22/2021 • 3 minutes to read • Edit Online

The arguments in a function call have this form:

> *expression* (*expression-list*$_{opt}$) /* Function call */

In a function call, *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the arguments passed to the function. If the function takes no arguments, *expression-list* should contain the keyword `void`.

An argument can be any value with fundamental, structure, union, or pointer type. All arguments are passed by value. This means a copy of the argument is assigned to the corresponding parameter. The function does not know the actual memory location of the argument passed. The function uses this copy without affecting the variable from which it was originally derived.

Although you cannot pass arrays or functions as arguments, you can pass pointers to these items. Pointers provide a way for a function to access a value by reference. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The order in which arguments are evaluated can vary under different compilers and different optimization levels. However, the arguments and any side effects are completely evaluated before the function is entered. See Side Effects for information on side effects.

The *expression-list* in a function call is evaluated and the usual arithmetic conversions are performed on each argument in the function call. If a prototype is available, the resulting argument type is compared to the prototype's corresponding parameter. If they do not match, either a conversion is performed, or a diagnostic message is issued. The parameters also undergo the usual arithmetic conversions.

The number of expressions in *expression-list* must match the number of parameters, unless the function's prototype or definition explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the list of parameters and converts them, if necessary, as described above. See Calls with a Variable Number of Arguments for more information.

If the prototype's parameter list contains only the keyword `void`, the compiler expects zero arguments in the function call and zero parameters in the definition. A diagnostic message is issued if it finds any arguments.

Example

This example uses pointers as arguments:

```
int main()
{
    /* Function prototype */

    void swap( int *num1, int *num2 );
    int x, y;
    .
    .
    .
    swap( &x, &y );  /* Function call */
}

/* Function definition */

void swap( int *num1, int *num2 )
{
    int t;

    t = *num1;
    *num1 = *num2;
    *num2 = t;
}
```

In this example, the `swap` function is declared in `main` to have two arguments, represented respectively by identifiers `num1` and `num2`, both of which are pointers to `int` values. The parameters `num1` and `num2` in the prototype-style definition are also declared as pointers to `int` type values.

In the function call

```
swap( &x, &y )
```

the address of `x` is stored in `num1` and the address of `y` is stored in `num2`. Now two names, or "aliases," exist for the same location. References to `*num1` and `*num2` in `swap` are effectively references to `x` and `y` in `main`. The assignments within `swap` actually exchange the contents of `x` and `y`. Therefore, no `return` statement is necessary.

The compiler performs type checking on the arguments to `swap` because the prototype of `swap` includes argument types for each parameter. The identifiers within the parentheses of the prototype and definition can be the same or different. What is important is that the types of the arguments match those of the parameter lists in both the prototype and the definition.

See also

Function Calls

Calls with a Variable Number of Arguments

12/22/2021 • 2 minutes to read • Edit Online

A partial parameter list can be terminated by the ellipsis notation, a comma followed by three periods (, ...), to indicate that there may be more arguments passed to the function, but no more information is given about them. Type checking is not performed on such arguments. At least one parameter must precede the ellipsis notation and the ellipsis notation must be the last token in the parameter list. Without the ellipsis notation, the behavior of a function is undefined if it receives parameters in addition to those declared in the parameter list.

To call a function with a variable number of arguments, simply specify any number of arguments in the function call. An example is the `printf` function from the C run-time library. The function call must include one argument for each type name declared in the parameter list or the list of argument types.

All the arguments specified in the function call are placed on the stack unless the `__fastcall` calling convention is specified. The number of parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the parameters. You are responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. The STDARG.H file contains ANSI-style macros for accessing arguments of functions which take a variable number of arguments. Also, the XENIX-style macros in VARARGS.H are still supported.

This sample declaration is for a function that calls a variable number of arguments:

```
int average( int first, ...);
```

See also

Function Calls

Recursive Functions

12/22/2021 • 2 minutes to read • Edit Online

Any function in a C program can be called recursively; that is, it can call itself. The number of recursive calls is limited to the size of the stack. See the `/STACK` (Stack Allocations) linker option for information about linker options that set stack size. Each time the function is called, new storage is allocated for the parameters and for the `auto` and `register` variables so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with `static` storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Example

This example illustrates recursive calls:

```
int factorial( int num );      /* Function prototype */

int main()
{
    int result, number;
    .
    .
    .
    result = factorial( number );
}

int factorial( int num )      /* Function definition */
{
    .
    .
    .
    if ( ( num > 0 ) || ( num <= 10 ) )
        return( num * factorial( num - 1 ) );
}
```

See also

Function Calls

C Language Syntax Summary

12/22/2021 • 2 minutes to read • Edit Online

This section gives the full description of the C language and the Microsoft-specific C language features. You can use the syntax notation in this section to determine the exact syntax for any language component. The explanation for the syntax appears in the section of this manual where a topic is discussed.

NOTE

This syntax summary is not part of the ANSI C standard, but is included for information only. Microsoft-specific syntax is noted in comments following the syntax.

See also

C Language Reference

Definitions and Conventions

12/22/2021 • 2 minutes to read • Edit Online

Terminals are endpoints in a syntax definition. No other resolution is possible. Terminals include the set of reserved words and user-defined identifiers.

Nonterminals are placeholders in the syntax and are defined elsewhere in this syntax summary. Definitions can be recursive.

An optional component is indicated by the subscripted $_{\text{opt}}$. For example,

> { *expression*$_{\text{opt}}$ }

indicates an optional expression enclosed in braces.

The syntax conventions use different font attributes for different components of the syntax. The symbols and fonts are as follows:

ATTRIBUTE	DESCRIPTION
nonterminal	Italic type indicates nonterminals.
`const`	Terminals in bold type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case sensitive.
$_{\text{opt}}$	Nonterminals followed by $_{\text{opt}}$ are always optional.
default typeface	Characters in the set described or listed in this typeface can be used as terminals in C statements.

A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced with the words "one of."

See also

C Language Syntax Summary

C lexical grammar

12/22/2021 • 3 minutes to read • Edit Online

Tokens

`token` :

`keyword`
`identifier`
`constant`
`string-literal`
`punctuator`

`preprocessing-token` :

`header-name`
`identifier`
`pp-number`
`character-constant`
`string-literal`
`punctuator`
each non-whitespace character that can't be one of the above

Keywords

`keyword` : one of

`auto` `break` `case` `char` `const` `continue`
`default` `do` `double` `else` `enum` `extern`
`float` `for` `goto` `if` `inline` `int` `long`
`register` `restrict` `return` `short` `signed`
`sizeof` `static` `struct` `switch` `typedef` `union`
`unsigned` `void` `volatile` `while` `_Alignas`
`_Alignof` `_Atomic` `_Bool` `_Complex` `_Generic`
`_Imaginary` `_Noreturn` `_Static_assert`
`_Thread_local`

For a list of additional Microsoft-specific keywords, see C keywords.

Identifiers

`identifier` :

`identifier-nondigit`
`identifier` *`identifier-nondigit`*
`identifier` *`digit`*

`identifier-nondigit` :

`nondigit`
`universal-character-name`
other implementation-defined characters

`nondigit` : one of

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

`digit` : one of
`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

`universal-character-name` :
`\u` *`hex-quad`*
`\U` *`hex-quad`* *`hex-quad`*

`hex-quad` :
`hexadecimal-digit` *`hexadecimal-digit`* *`hexadecimal-digit`* *`hexadecimal-digit`*

Constants

`constant` :
`integer-constant`
`floating-constant`
`enumeration-constant`
`character-constant`

`integer-constant` :
`decimal-constant` *`integer-suffix`* $_{opt}$
`binary-constant` [1] *`integer-suffix`* $_{opt}$
`octal-constant` *`integer-suffix`* $_{opt}$
`hexadecimal-constant` *`integer-suffix`* $_{opt}$

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

`binary-constant` :[1]
`binary-prefix` *`binary-digit`*
`binary-constant` *`binary-digit`*

`binary-prefix` [1]: one of
`0b` `0B`

`binary-digit` [1]: one of
`0` `1`

`octal-constant` :
`0`
`octal-constant` *`octal-digit`*

`hexadecimal-constant` :
`hexadecimal-prefix` *`hexadecimal-digit`*
`hexadecimal-constant` *`hexadecimal-digit`*

`hexadecimal-prefix` : one of
`0x` `0X`

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

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

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

`integer-suffix` :
  `unsigned-suffix` `long-suffix` opt
  `unsigned-suffix` `long-long-suffix` opt
  `long-suffix` `unsigned-suffix` opt
  `long-long-suffix` `unsigned-suffix` opt

`unsigned-suffix` : one of
  `u` `U`

`long-suffix` : one of
  `l` `L`

`long-long-suffix` : one of
  `ll` `LL`

`floating-constant` :
  `decimal-floating-constant`
  `hexadecimal-floating-constant`

`decimal-floating-constant` :
  `fractional-constant` `exponent-part` opt `floating-suffix` opt
  `digit-sequence` `exponent-part` `floating-suffix` opt

`hexadecimal-floating-constant` :
  `hexadecimal-prefix` `hexadecimal-fractional-constant` `binary-exponent-part` opt `floating-suffix` opt
  `hexadecimal-prefix` `hexadecimal-digit-sequence` `binary-exponent-part` `floating-suffix` opt

`fractional-constant` :
  `digit-sequence` opt `.` `digit-sequence`
  `digit-sequence` `.`

`exponent-part` :
  `e` `sign` opt `digit-sequence`
  `E` `sign` opt `digit-sequence`

`sign` : one of
  `+` `-`

`digit-sequence` :
  `digit`
  `digit-sequence` `digit`

`hexadecimal-fractional-constant` :
  `hexadecimal-digit-sequence` opt `.` `hexadecimal-digit-sequence`
  `hexadecimal-digit-sequence` `.`

`binary-exponent-part` :
  `p` `sign` opt `digit-sequence`

`P` `sign`opt `digit-sequence`

`hexadecimal-digit-sequence` :
  `hexadecimal-digit`
  `hexadecimal-digit-sequence` `hexadecimal-digit`

`floating-suffix` : one of
  `f` `l` `F` `L`

`enumeration-constant` :
  `identifier`

`character-constant` :
  `'` `c-char-sequence` `'`
  `L'` `c-char-sequence` `'`

`c-char-sequence` :
  `c-char`
  `c-char-sequence` `c-char`

`c-char` :
  Any member of the source character set except the single quotation mark (`'`), backslash (`\`), or newline character
  `escape-sequence`

`escape-sequence` :
  `simple-escape-sequence`
  `octal-escape-sequence`
  `hexadecimal-escape-sequence`
  `universal-character-name`

`simple-escape-sequence` : one of
  `\a` `\b` `\f` `\n` `\r` `\t` `\v`
  `\'` `\"` `\\` `\?`

`octal-escape-sequence` :
  `\` `octal-digit`
  `\` `octal-digit` `octal-digit`
  `\` `octal-digit` `octal-digit` `octal-digit`

`hexadecimal-escape-sequence` :
  `\x` `hexadecimal-digit`
  `hexadecimal-escape-sequence` `hexadecimal-digit`

String literals

`string-literal` :
  `encoding-prefix` `s-char-sequence`opt `"`

`encoding-prefix` :
  `u8`
  `u`
  `U`
  `L`

`s-char-sequence` :

`s-char`
`s-char-sequence` `s-char`

`s-char` :
any member of the source character set except the double-quotation mark (`"`), backslash (`\`), or newline character
`escape-sequence`

Punctuators

`punctuator` : one of
`[` `]` `(` `)` `{` `}` `.` `->`
`++` `--` `&` `*` `+` `-` `~` `!`
`/` `%` `<<` `>>` `<` `>` `<=` `>=` `==`
`!=` `^` `|` `&&` `||` `?` `:` `;` `...`
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=`
`&=` `^=` `|=` `,` `#` `##`
`<:` `:>` `<%` `%>` `%:` `%:%:`

Header names

`header-name` :
`<` `h-char-sequence` `>`
`"` `q-char-sequence` `"`

`h-char-sequence` :
`h-char`
`h-char-sequence` `h-char`

`h-char` :
any member of the source character set except the new-line character and `>`

`q-char-sequence` :
`q-char`
`q-char-sequence` `q-char`

`q-char` :
any member of the source character set except the new-line character and `"`

Preprocessing numbers

`pp-number` :
`digit`
`.` `digit`
`pp-number` `digit`
`pp-number` `identifier-nondigit`
`pp-number` `e` `sign`
`pp-number` `E` `sign`
`pp-number` `p` `sign`
`pp-number` `P` `sign`
`pp-number` `.`

[1] `binary-constant` , `binary-prefix` , and `binary-digit` are Microsoft-specific extensions.

See also

C Language Syntax Summary

Phrase Structure Grammar

12/22/2021 • 2 minutes to read • Edit Online

- Expressions
- Declarations
- Statements
- External Definitions

See also

C Language Syntax Summary

Summary of Expressions

12/22/2021 • 2 minutes to read • Edit Online

`primary-expression` :
  `identifier`
  `constant`
  `string-literal`
  `(` `expression` `)`
  `generic-selection`

`generic-selection` :
  `_Generic` `(` `assignment-expression` `,` `generic-assoc-list` `)`

`generic-assoc-list` :
  `generic-association`
  `generic-assoc-list` `,` `generic-association`

`generic-association` :
  `type-name` `:` `assignment-expression`
  `default` `:` `assignment-expression`

`postfix-expression` :
  `primary-expression`
  `postfix-expression` `[` `expression` `]`
  `postfix-expression` `(` `argument-expression-list` opt `)`
  `postfix-expression` `.` `identifier`
  `postfix-expression` `->` `identifier`
  `postfix-expression` `++`
  `postfix-expression` `--`
  `(` `type-name` `)` `{` `initializer-list` `}`
  `(` `type-name` `)` `{` `initializer-list` `,` `}`

`argument-expression-list` :
  `assignment-expression`
  `argument-expression-list` `,` `assignment-expression`

`unary-expression` :
  `postfix-expression`
  `++` `unary-expression`
  `--` `unary-expression`
  `unary-operator` `cast-expression`
  `sizeof` `unary-expression`
  `sizeof` `(` `type-name` `)` `_Alignof` `(` `type-name` `)`

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

`cast-expression` :
  `unary-expression`
  `(` `type-name` `)` `cast-expression`

multiplicative-expression:
  cast-expression
  multiplicative-expression `*` *cast-expression*
  multiplicative-expression `/` *cast-expression*
  multiplicative-expression `%` *cast-expression*

additive-expression:
  multiplicative-expression
  additive-expression `+` *multiplicative-expression*
  additive-expression `-` *multiplicative-expression*

shift-expression:
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

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

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

AND-expression:
  equality-expression
  AND-expression `&` *equality-expression*

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression `^` *AND-expression*

inclusive-OR-expression:
  exclusive-OR-expression
  inclusive-OR-expression `|` *exclusive-OR-expression*

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

logical-OR-expression:
  logical-AND-expression
  logical-OR-expression `||` *logical-AND-expression*

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

assignment-expression:
  conditional-expression
  unary-expression *assignment-operator* *assignment-expression*

`assignment-operator` : one of
  `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

`expression` :
  `assignment-expression`
  `expression` `,` `assignment-expression`

`constant-expression` :
  `conditional-expression`

See also

- Phrase Structure Grammar

Summary of Declarations

12/22/2021 • 2 minutes to read • Edit Online

declaration :
  declaration-specifiers *attribute-seq* opt [1] *init-declarator-list* opt `;`
  static_assert-declaration

declaration-specifiers :
  storage-class-specifier *declaration-specifiers* opt
  type-specifier *declaration-specifiers* opt
  type-qualifier *declaration-specifiers* opt
  function-specifier *declaration-specifiers* opt
  alignment-specifier *declaration-specifiers* opt

attribute-seq [1]:
  attribute [1] *attribute-seq* opt [1]

attribute [1, 2]: one of
  `__asm` `__based` `__cdecl` `__clrcall` `__fastcall` `__inline` `__stdcall` `__thiscall` `__vectorcall`

init-declarator-list :
  init-declarator
  init-declarator-list `,` *init-declarator*

init-declarator :
  declarator
  declarator `=` *initializer*

storage-class-specifier :
  `auto`
  `extern`
  `register`
  `static`
  `_Thread_local`
  `typedef`
  `__declspec` `(` *extended-decl-modifier-seq* `)` [1]

extended-decl-modifier-seq [1]:
  extended-decl-modifier opt
  extended-decl-modifier-seq *extended-decl-modifier*

extended-decl-modifier [1]:
  `thread`
  `naked`
  `dllimport`
  `dllexport`

type-specifier :
  `void`
  `char`
  `short`

`int`

`__int8` [1]

`__int16` [1]

`__int32` [1]

`__int64` [1]

`long`

`float`

`double`

`signed`

`unsigned`

`_Bool`

`_Complex`

`atomic-type-specifier`

`struct-or-union-specifier`

`enum-specifier`

`typedef-name`

`struct-or-union-specifier` :

`struct-or-union` *`identifier`* opt `{` *`struct-declaration-list`* `}`

`struct-or-union` *`identifier`*

`struct-or-union` :

`struct`

`union`

`struct-declaration-list` :

`struct-declaration`

`struct-declaration-list` *`struct-declaration`*

`struct-declaration` :

`specifier-qualifier-list` *`struct-declarator-list`* opt `;`

`static_assert-declaration`

`specifier-qualifier-list` :

`type-specifier` *`specifier-qualifier-list`* opt

`type-qualifier` *`specifier-qualifier-list`* opt

`alignment-specifier` *`specifier-qualifier-list`* opt

`struct-declarator-list` :

`struct-declarator`

`struct-declarator-list` `,` *`struct-declarator`*

`struct-declarator` :

`declarator`

`declarator` opt `:` *`constant-expression`*

`enum-specifier` :

`enum` *`identifier`* opt `{` *`enumerator-list`* `}`

`enum` *`identifier`* opt `{` *`enumerator-list`* `,` `}`

`enum` *`identifier`*

`enumerator-list` :

`enumerator`

`enumerator-list` `,` *`enumerator`*

`enumerator`:

  `enumeration-constant`
  `enumeration-constant` `=` `constant-expression`

`atomic-type-specifier`:

  `_Atomic` `(` `type-name` `)`

`type-qualifier`:

  `const`
  `restrict`
  `volatile`
  `_Atomic`

`function-specifier`:

  `inline`
  `_Noreturn`

`alignment-specifier`:

  `_Alignas` `(` `type-name` `)`
  `_Alignas` `(` `constant-expression` `)`

`declarator`:

  `pointer`$_{opt}$ `direct-declarator`

`direct-declarator`:

  `identifier`
  `(` `declarator` `)`
  `direct-declarator` `[` `type-qualifier-list`$_{opt}$ `assignment-expression`$_{opt}$ `]`
  `direct-declarator` `[` `static` `type-qualifier-list`$_{opt}$ `assignment-expression` `]`
  `direct-declarator` `[` `type-qualifier-list` `static` `assignment-expression` `]`
  `direct-declarator` `[` `type-qualifier-list`$_{opt}$ `*` `]`
  `direct-declarator` `(` `parameter-type-list` `)`
  `direct-declarator` `(` `identifier-list`$_{opt}$ `)`[3]

`pointer`:

  `*` `type-qualifier-list`$_{opt}$
  `*` `type-qualifier-list`$_{opt}$ `pointer`

`type-qualifier-list`:

  `type-qualifier`
  `type-qualifier-list` `type-qualifier`

`parameter-type-list`:

  `parameter-list`
  `parameter-list` `,` `...`

`parameter-list`:

  `parameter-declaration`
  `parameter-list` `,` `parameter-declaration`

`parameter-declaration`:

  `declaration-specifiers` `declarator`
  `declaration-specifiers` `abstract-declarator`$_{opt}$

`identifier-list`: /* For old-style declarator */

identifier
identifier-list `,` *identifier*

type-name :
specifier-qualifier-list *abstract-declarator* opt

abstract-declarator :
pointer
pointer opt *direct-abstract-declarator*

direct-abstract-declarator :
`(` *abstract-declarator* `)`
direct-abstract-declarator `[` *type-qualifier-list* opt *assignment-expression* opt `]`
direct-abstract-declarator `[` `static` *type-qualifier-list* opt *assignment-expression* `]`
direct-abstract-declarator `[` *type-qualifier-list* `static` *assignment-expression* `]`
direct-abstract-declarator `[` *type-qualifier-list* opt `*` `]`
direct-abstract-declarator opt `(` *parameter-type-list* opt `)`

typedef-name :
identifier

initializer :
assignment-expression
`{` *initializer-list* `}`
`{` *initializer-list* `, }`

initializer-list :
designation opt *initializer*
initializer-list `,` *designation* opt *initializer*

designation :
designator-list `=`

designator-list :
designator
designator-list *designator*

designator :
`[` *constant-expression* `]`
`.` *identifier*

static_assert-declaration :
`_Static_assert` `(` *constant-expression* `,` *string-literal* `)` `;`

[1] This grammar element is Microsoft-specific.

[2] For more information about these elements, see `__asm`, `__clrcall`, `__stdcall`, `__based`, `__fastcall`, `__thiscall`, `__cdecl`, `__inline`, and `__vectorcall`. [3] This style is obsolete.

See also

Calling conventions
Phrase structure grammar
Obsolete calling conventions

Summary of C statements

12/22/2021 • 2 minutes to read • Edit Online

statement :
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  try-except-statement /* Microsoft-specific */
  try-finally-statement /* Microsoft-specific */

jump-statement :
  `goto` *identifier* `;`
  `continue ;`
  `break ;`
  `return` *expression*opt `;`
  `__leave ;` /* Microsoft-specific[1] */

compound-statement :
  `{` *declaration-list*opt *statement-list*opt `}`

declaration-list :
  declaration
  declaration-list *declaration*

statement-list :
  statement
  statement-list *statement*

expression-statement :
  *expression*opt `;`

iteration-statement :
  `while (` *expression* `)` *statement*
  `do` *statement* `while (` *expression* `) ;`
  `for (` *expression*opt `;` *expression*opt `;` *expression*opt `)` *statement*

selection-statement :
  `if (` *expression* `)` *statement*
  `if (` *expression* `)` *statement* `else` *statement*
  `switch (` *expression* `)` *statement*

labeled-statement :
  identifier `:` *statement*
  `case` *constant-expression* `:` *statement*
  `default :` *statement*

try-except-statement : /* Microsoft-specific */
  `__try` *compound-statement* `__except (` *expression* `)` *compound-statement*

`try-finally-statement` : /* Microsoft-specific */
    `__try` `compound-statement` `__finally` `compound-statement`

1 The `__leave` keyword is only valid within the `__try` block of a `try-except-statement` or a `try-finally-statement`.

See also

Phrase structure grammar

External Definitions

12/22/2021 • 2 minutes to read • Edit Online

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration: /* Allowed only at external (file) scope */
  function-definition
  declaration

function-definition: /* Declarator here is the function declarator */
  declaration-specifiers$_{\text{opt}}$ *declarator declaration-list*$_{\text{opt}}$ *compound-statement*

See also

Phrase Structure Grammar

Implementation-Defined Behavior

12/22/2021 • 2 minutes to read • Edit Online

ANSI X3.159-1989, *American National Standard for Information Systems - Programming Language - C*, contains a section called "Portability Issues." The ANSI section lists areas of the C language that ANSI leaves open to each particular implementation. This section describes how Microsoft C handles these implementation-defined areas of the C language.

This section follows the same order as the ANSI section. Each item covered includes references to the ANSI that explains the implementation-defined behavior.

> **NOTE**
>
> This section describes the U.S. English-language version of the C compiler only. Implementations of Microsoft C for other languages may differ slightly.

See also

C Language Reference

Translation: Diagnostics

12/22/2021 • 2 minutes to read • Edit Online

ANSI 2.1.1.3 How a diagnostic is identified

Microsoft C produces error messages in the form:

> *filename* **(** *line-number* **)** **:** *diagnostic* **C** *number message*

where *filename* is the name of the source file in which the error was encountered; *line-number* is the line number at which the compiler detected the error; *diagnostic* is either "error" or "warning"; *number* is a unique four-digit number (preceded by a **C**, as noted in the syntax) that identifies the error or warning; *message* is an explanatory message.

See also

Implementation-Defined Behavior

Environment

12/22/2021 • 2 minutes to read • Edit Online

- Arguments to main
- Interactive Devices

See also

Implementation-Defined Behavior

Arguments to main

12/22/2021 • 2 minutes to read • Edit Online

ANSI 2.1.2.2.1 The semantics of the arguments to main

In Microsoft C, the function called at program startup is called **main**. There is no prototype declared for **main**, and it can be defined with zero, two, or three parameters:

```
int main( void )
int main( int argc, char *argv[] )
int main( int argc, char *argv[], char *envp[] )
```

The third line above, where **main** accepts three parameters, is a Microsoft extension to the ANSI C standard. The third parameter, **envp**, is an array of pointers to environment variables. The **envp** array is terminated by a null pointer. See The main Function and Program Execution for more information about **main** and **envp**.

The variable **argc** never holds a negative value.

The array of strings ends with **argv[argc]**, which contains a null pointer.

All elements of the **argv** array are pointers to strings.

A program invoked with no command-line arguments will receive a value of one for **argc**, as the name of the executable file is placed in **argv[0]**. (In MS-DOS versions prior to 3.0, the executable-file name is not available. The letter "C" is placed in **argv[0]**.) Strings pointed to by **argv[1]** through **argv[argc - 1]** represent program parameters.

The parameters **argc** and **argv** are modifiable and retain their last-stored values between program startup and program termination.

See also

Environment

Interactive Devices

12/22/2021 • 2 minutes to read • Edit Online

ANSI 2.1.2.3 What constitutes an interactive device

Microsoft C defines the keyboard and the display as interactive devices.

See also

Environment

Behavior of Identifiers

12/22/2021 • 2 minutes to read • Edit Online

- Significant Characters Without External Linkage
- Significant Characters with External Linkage
- Uppercase and Lowercase

See also

Using extern to Specify Linkage

Significant Characters Without External Linkage

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.2 The number of significant characters without external linkage

Identifiers are significant to 247 characters. The compiler does not restrict the number of characters you can use in an identifier; it simply ignores any characters beyond the limit.

See also

Using extern to Specify Linkage

Significant Characters with External Linkage

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.2 The number of significant characters with external linkage

Identifiers declared `extern` in programs compiled with Microsoft C are significant to 247 characters. You can modify this default to a smaller number using the /H (restrict length of external names) option.

See also

Using extern to Specify Linkage

Uppercase and Lowercase

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.2 Whether case distinctions are significant

Microsoft C treats identifiers within a compilation unit as case sensitive.

The Microsoft linker is case sensitive. You must specify all identifiers consistently according to case.

See also

Behavior of Identifiers

Characters

12/22/2021 • 2 minutes to read • Edit Online

- The ASCII Character Set
- Multibyte Characters
- Bits per Character
- Character Sets
- Unrepresented Character Constants
- Wide Characters
- Converting Multibyte Characters
- Range of char Values

See also

Implementation-Defined Behavior

ASCII Character Set

12/22/2021 • 2 minutes to read • Edit Online

ANSI 2.2.1 Members of source and execution character sets

The source character set is the set of legal characters that can appear in source files. For Microsoft C, the source character set is the standard ASCII character set.

> **NOTE**
>
> **Warning** Because keyboard and console drivers can remap the character set, programs intended for international distribution should check the Country/Region code.

See also

Characters

Multibyte Characters

12/22/2021 • 2 minutes to read • Edit Online

ANSI 2.2.1.2 Shift states for multibyte characters

Multibyte characters are used by some implementations, including Microsoft C, to represent foreign-language characters not represented in the base character set. However, Microsoft C does not support any state-dependent encodings. Therefore, there are no shift states. See Multibyte and Wide Characters for more information.

See also

Characters

Bits per Character

12/22/2021 • 2 minutes to read • Edit Online

ANSI 2.2.4.2.1 Number of bits in a character

The number of bits in a character is represented by the manifest constant **CHAR_BIT**. The LIMITS.H file defines **CHAR_BIT** as 8.

See also

Characters

Character Sets

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.3.4 Mapping members of the source character set

The source character set and execution character set include the ASCII characters listed in the following table. Escape sequences are also shown in the table.

Escape Sequences

ESCAPE SEQUENCE	CHARACTER	ASCII VALUE
`\a`	Alert/bell	7
`\b`	Backspace	8
`\f`	Form feed	12
`\n`	Newline	10
`\r`	Carriage return	13
`\t`	Horizontal tab	9
`\v`	Vertical tab	11
`\"`	Double quotation	34
`\'`	Single quotation	39
`\\`	Backslash	92

See also

Characters

Unrepresented Character Constants

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant

All character constants or escape sequences can be represented in the extended character set.

See also

Characters

Wide Characters

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.3.4 The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character

The regular character constant, 'ab' has the integer value (int)0x6162. When there is more than one byte, previously read bytes are shifted left by the value of **CHAR_BIT** and the next byte is compared using the bitwise-OR operator with the low **CHAR_BIT** bits. The number of bytes in the multibyte character constant cannot exceed sizeof(int), which is 4 for 32-bit target code.

The multibyte character constant is read as above and this is converted to a wide-character constant using the `mbtowc` run-time function. If the result is not a valid wide-character constant, an error is issued. In any event, the number of bytes examined by the `mbtowc` function is limited to the value of `MB_CUR_MAX`.

See also

Characters

Converting Multibyte Characters

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant

The current locale is the "C" locale by default. It can be changed with the #pragma setlocale.

See also

Characters

Range of char Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.2.1.1 Whether a "plain" `char` has the same range of values as a `signed char` or an `unsigned char`

All signed character values range from -128 to 127. All unsigned character values range from 0 to 255.

The `/J` compiler option changes the default type for `char` from `signed char` to `unsigned char`.

See also

Characters

Integers

12/22/2021 • 2 minutes to read • Edit Online

See also

Implementation-Defined Behavior

Range of Integer Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.2.5 The representations and sets of values of the various types of integers

Integers contain 32 bits (four bytes). Signed integers are represented in two's-complement form. The most-significant bit holds the sign: 1 for negative, 0 for positive and zero. The values are listed below:

TYPE	MINIMUM AND MAXIMUM
`unsigned short`	0 to 65535
`signed short`	-32768 to 32767
`unsigned long`	0 to 4294967295
`signed long`	-2147483648 to 2147483647

See also

Integers

Demotion of Integers

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented

When a `long` integer is cast to a `short`, or a `short` is cast to a `char`, the least-significant bytes are retained.

For example, this line

```
short x = (short)0x12345678L;
```

assigns the value 0x5678 to `x`, and this line

```
char y = (char)0x1234;
```

assigns the value 0x34 to `y`.

When `signed` variables are converted to `unsigned` and vice-versa, the bit patterns remain the same. For example, casting -2 (0xFE) to an `unsigned` value yields 254 (also 0xFE).

See also

Integers

Signed Bitwise Operations

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.3 The results of bitwise operations on signed integers

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers. For example, `-16 & 99` can be expressed in binary as

```
  11111111 11110000
& 00000000 01100011
  _________________
  00000000 01100000
```

The result of the bitwise AND is 96.

See also

Integers

Remainders

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.3.5 The sign of the remainder on integer division

The sign of the remainder is the same as the sign of the dividend. For example,

```
50 / -6 == -8
50 % -6 ==  2
-50 /  6 == -8
-50 %  6 == -2
```

See also

Integers

Right Shifts

12/22/2021 • 2 minutes to read • Edit Online

The result of a right shift of a negative-value signed integral type

Shifting a negative value to the right yields half the absolute value, rounded down. For example, a `signed short` value of -253 (hex 0xFF03, binary 11111111 00000011) shifted right one bit produces -127 (hex 0xFF81, binary 11111111 10000001). A positive 253 shifted right produces +126.

Right shifts preserve the sign bit of signed integral types. When a signed integer shifts right, the most-significant bit remains set. For example, if 0xF0000000 is a signed `int`, a right shift produces 0xF8000000. Shifting a negative `int` right 32 times produces 0xFFFFFFFF.

When an unsigned integer shifts right, the most-significant bit is cleared. For example, if 0xF000 is unsigned, the result is 0x7800. Shifting an `unsigned` or positive `int` right 32 times produces 0x00000000.

See also

Integers

Floating-Point Math

12/22/2021 • 2 minutes to read • Edit Online

See also

Implementation-Defined Behavior

Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.1.2.5 The representations and sets of values of the various types of floating-point numbers

The `float` type contains 32 bits: 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its range is +/- 3.4E38 with at least 7 digits of precision.

The `double` type contains 64 bits: 1 for the sign, 11 for the exponent, and 52 for the mantissa. Its range is +/- 1.7E308 with at least 15 digits of precision.

The `long double` type is distinct, but has the same representation as type `double` in the Microsoft C compiler.

See also

Floating-Point Math

Casting Integers to Floating-Point Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

For example, casting an `unsigned long` (with 32 bits of precision) to a `float` (whose mantissa has 23 bits of precision) rounds the number to the nearest multiple of 256. The `long` values 4,294,966,913 to 4,294,967,167 are all rounded to the `float` value 4,294,967,040.

See also

Floating-Point Math

Truncation of Floating-Point Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number

When an underflow occurs, the value of a floating-point variable is rounded to zero. An overflow may cause a run-time error or it may produce an unpredictable value, depending on the optimizations specified.

See also

Floating-Point Math

Arrays and Pointers

12/22/2021 • 2 minutes to read • Edit Online

- Largest Array Size
- Pointer Subtraction

See also

Implementation-Defined Behavior

Largest Array Size

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.3.3.4, 4.1.1 The type of integer required to hold the maximum size of an array — that is, the size of **size_t**

The `size_t` typedef is an `unsigned int` on the 32-bit x86 platform. On 64-bit platforms, the `size_t` typedef is an `unsigned __int64`.

See also

Arrays and Pointers

Pointer Subtraction

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.3.6, 4.1.1 The type of integer required to hold the difference between two pointers to elements of the same array, **ptrdiff_t**

The `ptrdiff_t` typedef is an `int` on the 32-bit x86 platform. On 64-bit platforms, the `ptrdiff_t` typedef is an `__int64`.

See also

Arrays and Pointers

Registers: Availability of Registers

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.1 The extent to which objects can actually be placed in registers by use of the register storage-class specifier

The compiler does not honor user requests for register variables. Instead, it makes it own choices when optimizing.

See also

Implementation-Defined Behavior

Structures, Unions, Enumerations, and Bit Fields

12/22/2021 • 2 minutes to read • Edit Online

- Improper Access to a Union
- Padding and Alignment of Structure Members
- Sign of Bit Fields
- Storage of Bit Fields
- The enum Type

See also

Implementation-Defined Behavior

Improper Access to a Union

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.3.2.3 A member of a union object is accessed using a member of a different type

If a union of two types is declared and one value is stored, but the union is accessed with the other type, the results are unreliable.

For example, a union of `float` and `int` is declared. A `float` value is stored, but the program later accesses the value as an `int`. In such a situation, the value would depend on the internal storage of `float` values. The integer value would not be reliable.

See also

Structures, Unions, Enumerations, and Bit Fields

Padding and Alignment of Structure Members

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.2.1 The padding and alignment of members of structures and whether a bit field can straddle a storage-unit boundary

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.

Every data object has an alignment-requirement. The alignment-requirement for all data except structures, unions, and arrays is either the size of the object or the current packing size (specified with either /Zp or the `pack` pragma, whichever is less). For structures, unions, and arrays, the alignment-requirement is the largest alignment-requirement of its members. Every object is allocated an offset so that

offset **%** *alignment-requirement* **==** 0

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit fields.

See also

Structures, Unions, Enumerations, and Bit Fields

Sign of Bit Fields

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.2.1 Whether a "plain" `int` field is treated as a `signed int` bit field or as an unsigned int bit field

Bit fields can be signed or unsigned. Plain bit fields are treated as signed.

See also

Structures, Unions, Enumerations, and Bit Fields

Storage of Bit Fields

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.2.1 The order of allocation of bit fields within an int

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

```
struct mybitfields
{
   unsigned a : 4;
   unsigned b : 5;
   unsigned c : 7;
} test;

int main( void )
{
   test.a = 2;
   test.b = 31;
   test.c = 0;
}
```

the bits would be arranged as follows:

```
00000001 11110010
cccccccb bbbbaaaa
```

Since the 80x86 processors store the low byte of integer values before the high byte, the integer 0x01F2 above would be stored in physical memory as 0xF2 followed by 0x01.

See also

Structures, Unions, Enumerations, and Bit Fields

enum Type

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.2.2 The integer type chosen to represent the values of an enumeration type

A variable declared as `enum` is an `int`.

See also

Structures, Unions, Enumerations, and Bit Fields

Qualifiers: Access to Volatile Objects

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.5.3 What constitutes an access to an object that has volatile-qualified type

Any reference to a volatile-qualified type is an access.

See also

Implementation-Defined Behavior

Declarators: Maximum number

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type

Microsoft C does not limit the number of declarators. The number is limited only by available memory.

See also

Implementation-Defined Behavior

Statements: Limits on Switch Statements

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.6.4.2 The maximum number of `case` values in a `switch` statement

Microsoft C does not limit the number of `case` values in a `switch` statement. The number is limited only by available memory.

See also

Implementation-Defined Behavior

Preprocessing Directives

12/22/2021 • 2 minutes to read • Edit Online

- Character Constants and Conditional Inclusion
- Including Bracketed Filenames
- Including Quoted Filenames
- Character Sequences
- Pragmas
- Default Date and Time

See also

Implementation-Defined Behavior

Character Constants and Conditional Inclusion

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value

The character set used in preprocessor statements is the same as the execution character set. The preprocessor recognizes negative character values.

See also

Preprocessing Directives

Including Bracketed Filenames

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.8.2 The method for locating includable source files

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A "parent" file is the file that has the #include directive in it. Instead, it begins by searching for the file in the directories specified on the compiler command line following /I. If the /I option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. The INCLUDE environment variable can contain multiple paths separated by semicolons (;). If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order in which they appear.

See also

Preprocessing Directives

Including Quoted Filenames

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.8.2 The support for quoted names for includable source files

If you specify a complete, unambiguous path specification for the include file between two sets of double quotation marks (" "), the preprocessor searches only that path specification and ignores the standard directories.

For include files specified as #include "path-spec", directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

See also

Preprocessing Directives

Character Sequences

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.8.2 The mapping of source file character sequences

Preprocessor statements use the same character set as source file statements with the exception that escape sequences are not supported.

Thus, to specify a path for an include file, use only one backslash:

```
#include "path1\path2\myfile"
```

Within source code, two backslashes are necessary:

```
fil = fopen( "path1\\path2\\myfile", "rt" );
```

See also

Preprocessing Directives

Pragmas

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.8.6 The behavior on each recognized #pragma directive.

The following C Pragmas are defined for the Microsoft C compiler:

alloc_text
auto_inline
check_stack
code_seg
comment

data_seg
function
hdrstop
include_alias

inline_depth
inline_recursion
intrinsic
message

optimize
pack
setlocale
warning

See also

Preprocessing Directives

Default Date and Time

12/22/2021 • 2 minutes to read • Edit Online

ANSI 3.8.8 The definitions for `__DATE__` and `__TIME__` when, respectively, the date and time of translation are not available

When the operating system does not provide the date and time of translation, the default values for `__DATE__` and `__TIME__` are `May 03 1957` and `17:00:00`.

See also

Preprocessing Directives

Library Functions

12/22/2021 • 2 minutes to read • Edit Online

- NULL Macro
- Diagnostic Printed by the assert Function
- Character Testing
- Domain Errors
- Underflow of Floating-Point Values
- The fmod Function
- The signal Function
- Default Signals
- Terminating Newline Characters
- Blank Lines
- Null Characters
- File Position in Append Mode
- Truncation of Text Files
- File Buffering
- Zero-Length Files
- Filenames
- File Access Limits
- Deleting Open Files
- Renaming with a Name That Exists
- Reading Pointer Values
- Reading Ranges
- File Position Errors
- Messages Generated by the perror Function
- Allocating Zero Memory
- The abort Function
- The atexit Function
- Environment Names
- The system Function
- The strerror Function

- The Time Zone
- The clock Function

See also

Implementation-Defined Behavior

NULL Macro

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.1.5 The null pointer constant to which the macro NULL expands

Several include files define the NULL macro as `((void *)0)`.

See also

Library Functions

Diagnostic Printed by the assert Function

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.2 The diagnostic printed by and the termination behavior of the **assert** function

The **assert** function prints a diagnostic message and calls the **abort** routine if the expression is false (0). The diagnostic message has the form

> **Assertion failed**: *expression*, **file** *filename*, **line** *linenumber*

where *filename* is the name of the source file and *linenumber* is the line number of the assertion that failed in the source file. No action is taken if *expression* is true (nonzero).

See also

Library Functions

Character Testing

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.3.1 The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions

The following list describes these functions as they are implemented by the Microsoft C compiler.

FUNCTION	TESTS FOR
`isalnum`	Characters 0 - 9, A-Z, a-z ASCII 48-57, 65-90, 97-122
`isalpha`	Characters A-Z, a-z ASCII 65-90, 97-122
`iscntrl`	ASCII 0 -31, 127
`islower`	Characters a-z ASCII 97-122
`isprint`	Characters A-Z, a-z, 0 - 9, punctuation, space ASCII 32-126
`isupper`	Characters A-Z ASCII 65-90

See also

Library Functions

Domain Errors

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.5.1 The values returned by the mathematics functions on domain errors

The ERRNO.H file defines the domain error constant `EDOM` as 33. See the help topic for the particular function that caused the error, for information about the return value.

See also

Library Functions

Underflow of Floating-Point Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.5.1 Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors

A floating-point underflow does not set the expression `errno` to `ERANGE`. When a value approaches zero and eventually underflows, the value is set to zero.

See also

Library Functions

fmod Function

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.5.6.4 Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero

When the `fmod` function has a second argument of zero, the function returns zero.

See also

Library Functions

signal Function (C)

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.7.1.1 The set of signals for the **signal** function

The first argument passed to **signal** must be one of the symbolic constants described in the *Run-Time Library Reference* for the **signal** function. The information in the *Run-Time Library Reference* also lists the operating mode support for each signal. The constants are also defined in SIGNAL.H.

See also

Library Functions

Default Signals

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.7.1.1 If the equivalent of `signal(sig, SIG_DFL)` is not executed prior to the call of a signal handler, the blocking of the signal that is performed

Signals are set to their default status when a program begins running.

See also

Library Functions

Terminating Newline Characters

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.2 Whether the last line of a text stream requires a terminating newline character

Stream functions recognize either new line or end of file as the terminating character for a line.

See also

Library Functions

Blank Lines

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in

Space characters are preserved.

See also

Library Functions

Null Characters

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.2 The number of null characters that can be appended to data written to a binary stream

Any number of null characters can be appended to a binary stream.

See also

Library Functions

File Position in Append Mode

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file

When a file is opened in append mode, the file-position indicator initially points to the end of the file.

See also

Library Functions

Truncation of Text Files

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point

Writing to a text stream does not truncate the file beyond that point.

See also

Library Functions

File Buffering

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.3 The characteristics of file buffering

Disk files accessed through standard I/O functions are fully buffered. By default, the buffer holds 512 bytes.

See also

Library Functions

Zero-Length Files

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.3 Whether a zero-length file actually exists

Files with a length of zero are permitted.

See also

Library Functions

Filenames

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.3 The rules for composing valid file names

A file specification can include an optional drive letter (always followed by a colon), a series of optional directory names (separated by backslashes), and a filename.

For more information, see Naming a File for more information.

See also

Library Functions

File Access Limits

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.3 Whether the same file can be open multiple times

Opening a file that is already open is not permitted.

See also

Library Functions

Deleting Open Files

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.4.1 The effect of the remove function on an open file

The remove function deletes a file. If the file is open, this function fails and returns -1.

See also

Library Functions

Renaming with a Name That Exists

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.4.2 The effect if a file with the new name exists prior to a call to the **rename** function

If you attempt to rename a file using a name that exists, the **rename** function fails and returns an error code.

See also

Library Functions

Reading Pointer Values

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.6.2 The input for **%p** conversion in the `fscanf` function

When the **%p** format character is specified, the `fscanf` function converts pointers from hexadecimal ASCII values into the correct address.

See also

Library Functions

Reading Ranges

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.6.2 The interpretation of a dash (-) character that is neither the first nor the last character in the scanlist for % [conversion in the `fscanf` function

The following line

```
fscanf( fileptr, "%[A-Z]", strptr);
```

reads any number of characters in the range A-Z into the string to which `strptr` points.

See also

Library Functions

File Position Errors

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.9.1, 4.9.9.4 The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure

When `fgetpos` or `ftell` fails, `errno` is set to the manifest constant `EINVAL` if the position is invalid or `EBADF` if the file number is bad. The constants are defined in ERRNO.H.

See also

Library Functions

Messages Generated by the perror Function

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.9.10.4 The messages generated by the `perror` function

The `perror` function generates these messages:

```
0  Error 0
1
2  No such file or directory
3
4
5
6
7  Arg list too long
8  Exec format error
9  Bad file number
10
11
12 Not enough core
13 Permission denied
14
15
16
17 File exists
18 Cross-device link
19
20
21
22 Invalid argument
23
24 Too many open files
25
26
27
28 No space left on device
29
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur
```

See also

Library Functions

Allocating Zero Memory

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.10.3 The behavior of the `calloc`, `malloc`, or `realloc` function if the size requested is zero

The `calloc`, `malloc`, and `realloc` functions accept zero as an argument. No actual memory is allocated, but a valid pointer is returned and the memory block can be modified later by realloc.

See also

Library Functions

abort Function (C)

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.10.4.1 The behavior of the `abort` function with regard to open and temporary files

The `abort` function does not close files that are open or temporary. It does not flush stream buffers. For more information, see abort.

See also

Library Functions

atexit Function (C)

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.10.4.3 The status returned by the `atexit` function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE

The `atexit` function returns zero if successful, or a nonzero value if unsuccessful.

See also

Library Functions

Environment Names

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.10.4.4 The set of environment names and the method for altering the environment list used by the getenv function

The set of environment names is unlimited.

To change environment variables from within a C program, call the _putenv function. To change environment variables from the command line in Windows, use the SET command (for example, SET LIB = D:\ LIBS).

Environment variables set from within a C program exist only as long as their host copy of the operating system command shell is running (CMD.EXE or COMMAND.COM). For example, the line

```
system( SET LIB = D:\LIBS );
```

would run a copy of the command shell (CMD.EXE), set the environment variable LIB, and return to the C program, exiting the secondary copy of CMD.EXE. Exiting that copy of CMD.EXE removes the temporary environment variable LIB.

Likewise, changes made by the `_putenv` function last only until the program ends.

See also

Library Functions
_putenv, _wputenv
getenv, _wgetenv

system Function

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.10.4.5 The contents and mode of execution of the string by the **system** function

The **system** function executes an internal operating system command, or an .EXE, .COM (.CMD in Windows NT) or .BAT file from within a C program rather than from the command line.

The system function finds the command interpreter, which is typically CMD.EXE in the Windows NT operating system or COMMAND.COM in Windows. The system function then passes the argument string to the command interpreter.

For more information, see system, _wsystem.

See also

Library Functions

strerror Function

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.11.6.2 The contents of the error message strings returned by the `strerror` function

The `strerror` function generates these messages:

```
0   Error 0
1
2   No such file or directory
3
4
5
6
7   Arg list too long
8   Exec format error
9   Bad file number
10
11
12  Not enough core
13  Permission denied
14
15
16
17  File exists
18  Cross-device link
19
20
21
22  Invalid argument
23
24  Too many open files
25
26
27
28  No space left on device
29
30
31
32
33  Math argument
34  Result too large
35
36  Resource deadlock would occur
```

See also

[Library Functions]

Time Zone

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.12.1 The local time zone and Daylight Saving Time

The local time zone is Pacific Standard Time. Microsoft C supports Daylight Saving Time.

See also

Library Functions

clock Function (C)

12/22/2021 • 2 minutes to read • Edit Online

ANSI 4.12.2.1 The era for the `clock` function

The `clock` function's era begins (with a value of 0) when the C program starts to execute. It returns times measured in 1/**CLOCKS_PER_SEC** (which equals 1/1000 for Microsoft C).

See also

Library Functions

C/C++ preprocessor reference

12/22/2021 • 2 minutes to read • Edit Online

The *C/C++ preprocessor reference* explains the preprocessor as it is implemented in Microsoft C/C++. The preprocessor performs preliminary operations on C and C++ files before they are passed to the compiler. You can use the preprocessor to conditionally compile code, insert files, specify compile-time error messages, and apply machine-specific rules to sections of code.

In Visual Studio 2019 the /Zc:preprocessor compiler option provides a fully conformant C11 and C17 preprocessor. This is the default when you use the compiler flag `/std:c11` or `/std:c17`.

In this section

Preprocessor
Provides an overview of the traditional and new conforming preprocessors.

Preprocessor directives
Describes directives, typically used to make source programs easy to change and easy to compile in different execution environments.

Preprocessor operators
Discusses the four preprocessor-specific operators used in the context of the `#define` directive.

Predefined macros
Discusses predefined macros as specified by the C and C++ standards and by Microsoft C++.

Pragmas
Discusses pragmas, which offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

Related sections

C++ language reference
Provides reference material for the Microsoft implementation of the C++ language.

C language reference
Provides reference material for the Microsoft implementation of the C language.

C/C++ build reference
Provides links to topics discussing compiler and linker options.

Visual Studio projects - C++
Describes the user interface in Visual Studio that enables you to specify the directories that the project system will search to locate files for your C++ project.

Microsoft C runtime library (CRT) reference

12/22/2021 • 2 minutes to read • Edit Online

The Microsoft runtime library provides routines for programming the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages.

Sample programs are included in the individual reference topics for most routines in the library.

In This Section

Universal C runtime routines by category
Provides links to the runtime library by category.

Global variables and standard types
Provides links to the global variables and standard types provided by the runtime library.

Global constants
Provides links to the global constants defined by the runtime library.

Global state
Describes the scope of global state in the C runtime library.

Generic-text mappings
Provides links to the generic-text mappings defined in Tchar.h.

Alphabetical function reference
Provides links to the C runtime library functions, organized alphabetically.

Function family overviews
Provides links to the C runtime library functions, organized by function family.

Language and country/region strings
Describes how to use the `setlocale` function to set the language and Country/Region strings.

C runtime (CRT) and C++ Standard Library (STL) `.lib` files
List of `.lib` files that make up the C runtime libraries and their associated compiler options and preprocessor directives.

Related Sections

Debug routines
Provides links to the debug versions of the runtime library routines.

Runtime error checking
Provides links to functions that support runtime error checks.

DLLs and Visual C++ runtime library behavior
Discusses the entry point and startup code used for a DLL.

Debugging
Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.