! to ~

! — Operator

Logical negation operator
!*operand*

The operator **!** is the logical negation operator. Its operand must be an expression with scalar type. **!** then inverts the logical result of its operand. This result has type **int**.

If *operand* is nonzero, **!***operand* yields zero; if *operand* is zero, then **!***operand* yields one.

The expression **!***operand* is equivalent to **(0==***operand***)**.

Cross-references

Standard, §3.3.3.3
The C Programming Language, ed. 2, p. 204

See Also

!=, ~, expressions

!= — Operator

Inequality operator
operand1 **!=** *operand2*

The operator **!=** compares *operand1* with *operand2*. The result of this operation is one if the operands are *not* equal, and zero if they are.

The operands must be one of the following:

- Arithmetic types.
- Pointers to compatible types (ignoring qualifiers on these types).
- A pointer to an object or incomplete type, and a pointer to **void**.
- A pointer and NULL.

If both operands have arithmetic type, they undergo usual arithmetic conversion before being compared. If one operand is a pointer to an object and the other is a pointer to **void**, the pointer to an object is converted to a pointer to **void** for purposes of the comparison.

Cross-references

Standard, §3.3.9
The C Programming Language, ed. 2, pp. 41, 207

See Also

!, ==, expressions

” — Punctuator

String literal character

The quotation mark “” marks the beginning and end of a string literal. To embed a quotation mark within a string literal, use the escape sequence \".

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 194

See Also

string literal

—

String-ize operator

The operator **#** is read and translated by the preprocessor. It must be followed by one of the formal parameters of a function-like macro. The token sequence that would have replaced the formal parameter in the absence of the **#** is instead converted to a string literal, and the string literal replaces the both the **#** and the formal parameter. This process is called *string-izing*.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

The preprocessor replaced **#x** with a string literal that names the sequence of token that replaces **x**.

The following rules apply to interpreting the **#** operator:

1. If a sequence of white-space characters occurs within the preprocessing tokens that replace the argument, it is replaced with one space character.
2. All white-space characters that occur before the first preprocessing token and after the last preprocessing token is deleted.
3. The original spelling of the token that is stringized is retained in the string produced. This means that as the string is formed, the translator appropriately escapes any backslashes or quotation marks in the tokens.

Example

The following uses the operator **#** to display the result of several mathematics routines.

```
#include <errno.h>
#include <math.h>
#include <stddef.h>
#include <stdio.h>

void show(double value, char *name)
{
      if (errno)
            perror(name);
      else
            printf("%10g %s\n", value, name);
      errno = 0;
}

#define display(x) show((double)(x), #x)

main(void)
{
      extern char *gets();
      double x;
      char string[64];
```

```
      for(;;) {
            printf("Enter a number: ");
            fflush(stdout);
            if(gets(string) == NULL)
                  break;

            x = atof(string);
            display(x);
            display(cos(x));
            display(sin(x));
            display(tan(x));
            display(acos(cos(x)));
      }
}
```

Cross-references

Standard, §3.8.3.2
The C Programming Language, ed. 2, pp. 90, 230

See Also

##, #define, preprocessing

— Operator

Token-pasting operator

The operator **##** is is used by the preprocessor. It can be used in both object-like and function-like macros. When used immediately before or immediately after an element in the macro's replacement list, it joins the corresponding preprocessor token with its neighbor. This is sometimes called "token pasting".

As an example of token pasting, consider the macro:

```
#define printvar(number) printf("%s\n", variable ## number)
```

When the preprocessor reads the following line

```
printvar(5);
```

it substitutes the following code for it:

```
printf("%s\n", variable5);
```

The preprocessor throws away all white space both before and after the **##** operator.

The **##** operator must not be used as the first or last entry in a replacement list.

All instances of the **##** operator are resolved before further macro replacement is performed.

Cross-references

Standard, §3.8.3.3
The C Programming Language, ed. 2, pp. 90, 230

See Also

#, #define, preprocessing

Notes

Some pre-ANSI translators supported token pasting by replacing a comment in a macro replacement list with no space. ANSI translators always replace a comment with one space, no matter where that comment appears.

LEXICON

The order of evaluation of multiple **##** operators is unspecified.

#define — Preprocessing directive

Define an identifier as a macro
#define *identifier replacement-list*
#define *identifier* **(** *parameter-list*$_{opt}$ **)** *replacement-list*

The preprocessing directive **#define** tells the preprocessor to regard *identifier* as a macro.

#define can define two kinds of macros: *object-like*, and *function-like*.

Object-like Macros

An object-like macro has the syntax

```
#define identifier replacement-list
```

This type of macro is also called a *manifest constant*.

The preprocessor searches for *identifier* throughout the text of the translation unit, excluding comments, string literals, and character constants, and replaces it with the elements of *replacement-list*, which is then rescanned for further macro substitutions.

For example, consider the directive:

```
#define BUFFERSIZE 75
```

When the preprocessor reads the line

```
malloc(BUFFERSIZE);
```

it replaces it with:

```
malloc(75);
```

Function-like Macros

A function-like macro is more complex. The preprocessor looks for *identifier***(***argument-list***)** throughout the text of the translation unit, excluding comments, string literals, and character constants. The number of comma-separated arguments in *argument-list* must match the number of comma-separated parameters in the *parameter-list* of the macro's definition. The list is optional in the sense that some function-like macros do not have any parameters.

In the following description, *argument* means the sequence of tokens in *argument-list* that occupies the same relative position as the parameter under discussion occupies in *parameter-list*. The preprocessor replaces *identifier***(***argument-list***)** with the *replacement-list* specified in the definition after it performs the following substitutions: If a parameter is followed or preceded by the operator **##**, then the parameter is replaced by the argument. If a parameter is preceded by **#**, then the **#** and the parameter are replaced by a string literal that contains the argument. All other instances of parameters are replaced by the argument after the argument has first been exhaustively scanned for further preprocessor macro expansions. All instances of **##** are converted to token-paste operations.

For example, the consider the macro:

```
#define display(x) show((long)(x), #x)
```

When the preprocessor reads the following line

```
display(abs(-5));
```

it replaces it with the following:

```
show((long)(abs(-5)), "abs(-5)");
```

When an argument to a function-like macro contains no preprocessing tokens, or when an argument to a function-like macro contains a preprocessing token that is identical to a preprocessing directive, the behavior is undefined.

Macro Rescanning

As noted above, the preprocessor searches for macro identifiers throughout the text of the translation unit, excluding comments, string literals, and character constants. The text of replaced macros is also scanned for macro replacements, but it is not part of the text of the translation unit (i.e., source file), so it does not follow the same rules.

After it replaces the identifier of an object-like macro or the *identifier***(***argument-list***)** of a function-like macro with the appropriate *replacement-list*, the preprocessor continues to scan for further macro invocations, starting with the *replacement-list*.

While the preprocessor scans the *replacement-list*, it suppresses the definition of the macro that produced the list. If the preprocessor recognizes a second macro invocation and replaces it before it processes the tokens that replace the first invocation, then it suppresses the definitions of both the first and the second macros while it processes the *replacement-list* of the second macro.

The preprocessor suppresses a definition as long as any of the tokens that remain to be processed are derived directly from the original macro replacement or from further macro replacements that use parts of the original macro replacement. Thus, when the object-like macro definition

```
#define RECURSE     RE ## CURSE
```

is invoked by the token **RECURSE**, it is replaced by the token **RECURSE** formed by pasting **RE** and **CURSE** together, but the scanning of the replacement list would not invoke the macro RECURSE a second time. Likewise, the function-like macro definition

```
#define RECURSE(a, b)   a ## b(a, b)
```

when invoked with the sequence **RECURSE(RE, CURSE)** would be replaced by the token sequence **RECURSE(RE, CURSE)**, but the scanning of the replaced token sequence would not invoke the macro **RECURSE()** again.

Be warned that you should not test a PC-based compiler for compliance with these macro definitions unless you are prepared to turn off your machine. If the compiler fails to detect the recursion, it may become locked in an infinite loop, and there may be no other way to terminate the substitution.

Example

For an example of using a function-like macro in a program, see **#**.

Cross-references

Standard, §3.8.3
The C Programming Language, ed. 2, pp. 229*ff*

See Also

#, ##, #undef, preprocessing

Notes

A macro expansion always occupies exactly one line, no matter how many lines are spanned by the definition or the actual parameters.

A macro definition can extend over more than one line, provided that a backslash '\' appears before the newline character that breaks the lines. The size of a **#define** directive is therefore limited by the maximum size of a logical source line, which can be up to at least 509 characters long.

LEXICON

A macro may be redefined only if the new definition matches the old definition in all respects except the spelling of white space.

#elif — Preprocessing directive

Include code conditionally
#elif *constant-expression* <*newline*> *group*$_{opt}$

The preprocessing directive **#elif** conditionally includes code within a program. It can be used after any of the instructions **#if**, **#ifdef**, or **#ifndef**, and before **#endif** that ends the chain of conditional-inclusion directives.

If the conditional expression of the preceding **#if**, **#ifdef**, or **#ifndef** directive is false and the *constant-expression* that follows **#elif** is non-zero, then *group* is included within the program up to the next **#elif**, **#else**, or **#endif** directive. An **#if**, **#ifdef**, or **#ifndef** directive may be followed by any number of **#elif** directives.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation. The implementation defines whether the result of evaluating a character constant in *constant-expression* matches the result of evaluating the same character constant in a C expression. For example, it is up to the implementation whether

```
#elif 'z' - 'a' == 25
```

yields the same value as:

```
else if ('z' - 'a' == 25)
```

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#else, #endif, #if, #ifdef, #ifndef, preprocessing

#else — Preprocessing directive

Include code conditionally
#else *newline group*$_{opt}$

The preprocessing directive **#else** conditionally includes code within a program. It is preceded by one of the directives **#if**, **#ifdef**, or **#ifndef**, and may also be preceded by any number of **#elif** directives. If all preceding directives evaluate to false, then the code introduced by **#else** is included within the program up to the **#endif** directive that concludes the chain of conditional-inclusion directives.

A **#if**, **#ifdef**, or **#ifndef** directive can be followed by only one **#else** directive.

Example

For an example of using this directive in a program, see **assert**.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #endif, #if, #ifdef, #ifndef, preprocessing

#endif — Preprocessing directive

End conditional inclusion of code
#endif

The preprocessing directive **#endif** must follow any **#if**, **#ifdef**, or **#ifndef** directive. It may also be preceded by any number of **#elif** directives and an **#else** directive. It marks the end of a sequence of source-file statements that are included conditionally by the preprocessor.

Example

For an example of using this directive in a program, see **assert**.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #if, #ifdef, #ifndef, preprocessing

#error — Preprocessing directive

Error directive
#error *message newline*

The preprocessing directive **#error** prints *message* when an error occurs.

Cross-references

Standard, §3.8.5
The C Programming Language, ed. 2, p. 233

See Also

preprocessing

#if — Preprocessing directive

Include code conditionally
#if *constant-expression newline group*$_{opt}$

The preprocessing directive **#if** tells the preprocessor that if *constant-expression* is true, then include the following lines of code within the program until it reads the next **#elif**, **#else**, or **#endif** directive.

The *constant-expression* must be an integral expression, and it cannot include a **sizeof** operator, a cast, or an enumeration constant. All macro substitutions are performed upon the *constant-expression* before it is evaluated. All integer constants are treated as long objects, and are then evaluated. If *constant-expression* includes character constants, all escape sequences are converted into characters before evaluation.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #endif, #ifdef, #ifndef, preprocessing

LEXICON

Notes

The keyword **defined** determines whether a symbol is defined to **#if**. For example,

```
#if defined(SYMBOL)
```

or

```
#if defined SYMBOL
```

is equivalent to

```
#ifdef SYMBOL
```

except that it can be used in more complex expressions, such as

```
#if defined FOO && defined BAR && FOO==10
```

#ifdef — Preprocessing directive

Include code conditionally
#ifdef *identifier newline group*$_{opt}$

The preprocessing directive **#ifdef** checks whether *identifier* has been defined as a macro or manifest constant. If *identifier* has been defined, then the preprocessor includes *group* within the program, up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has not been defined, however, then *group* is skipped.

An **#ifdef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and must be followed by an **#endif** directive.

Example

For an example of using this directive in a program, see **assert**.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #endif, #if, #ifndef, defined, preprocessing

Notes

This is the same as:

```
#if defined IDENTIFIER
```

#ifndef — Preprocessing directive

Include code conditionally
#ifndef *identifier newline group*$_{opt}$

The preprocessing directive **#ifndef** checks whether *identifier* has been defined as a macro or manifest constant. If *identifier* has *not* been defined, then the preprocessor includes *group* within the program up to the next **#elif**, **#else**, or **#endif** directive. If *identifier* has been defined, however, then *group* is skipped.

An **#ifndef** directive can be followed by any number of **#elif** directives, by one **#else** directive, and by one **#elif** directive.

Cross-references

Standard, §3.8.1
The C Programming Language, ed. 2, p. 91

See Also

#elif, #else, #endif, #if, #ifndef, defined, preprocessing

Notes

This is the same as:

```
#if !defined IDENTIFIER
```

#include — Preprocessing directive

Read another file and include it
#include <*file***>**
#include "*file***"**

The preprocessing directive **#include** tells the preprocessor to replace the directive with the contents of *file*.

The directive can take one of two forms: either the name of the file is enclosed within angle brackets (**<***file***>**), or it is enclosed within quotation marks ("*file*"). The name of the file can be enclosed within angle brackets (**<***file***.h>**) or quotation marks ("*file***.h**"). Angle brackets tell the preprocessor to look for *file* in the directories named with the **-I** options to the **cc** command line, and then in the directory named by the environmental variable **INCDIR**. Quotation marks tell **cpp** to look for *file***.h** in the source file's directory, then in directories named with the **-I** options, and then in the directory named by the environmental variable **INCDIR**. **#include** directives may be nested up to at least eight deep. That is to say, a file included by an **#include** directive may use an **#include** directive to include a third file. That third file may also use a **#include** directive to include a fourth file, and so on, up to at least eight files.

A subordinate header is sought relative to the original source file, rather than relative to the header that calls it directly. For example, suppose that under the UNIX operating system, a file **example.c** resides in directory **/v/fred/src**. If **example.c** contains the directive **#include <header1.h>**. The operating system will look for **header1.h** in the standard directory, **/usr/include**. If **header1.h** includes the directive **#include <../header2.h>** then the implementation should look for **header2.h** not in directory **/usr**, but in directory **/v/fred/src**.

Some file systems allow characters to be used in file names that are used as delimiters in other file systems. Therefore, if any of the characters '*', "\', or ',' are part of a file name, behavior is undefined. If "" is part of a file name between angle-bracket delimiters, behavior is also undefined.

A **#include** directive may also take the form **#include** *string*, where *string* is a macro that expands into either of the two forms described above.

Cross-references

Standard, §2.2.4.1, §3.8.2
The C Programming Language, ed. 2, p. 88

See Also

header, header names, Language, preprocessing

Notes

Trigraphs that occur within a **#include** directive are substituted, because they are processed by an earlier phase of translation than are **#include** directives.

LEXICON

#line — Preprocessing directive

Reset line number
#line *number newline*
#line *number filename newline*
#line *macros newline*

#line is a preprocessing directive that resets the line number within a file. The Standard defines the line number as being the number of newline characters read, plus one.

#line can take any of three forms. The first, **#line** *number*, resets the current line number in the source file to *number*. The second, **#line** *number filename*, resets the line number to *number* and changes the name of the file referred to by _ _**FILE**_ _ to *filename*. The third, **#line** *macros*, contains macros that have been defined by earlier preprocessing directives. When the macros have been expanded by the preprocessor, the **#line** instruction will then resemble one of the first two forms and be interpreted appropriately.

number specifies the number of the next source line in the file, not the number of the **#line** directive's source line.

Cross-references

Standard, §3.8.4
The C Programming Language, ed. 2, p. 233

See Also

preprocessing

Notes

Most often, **#line** is used to ensure that error messages point to the correct line in the program's source code. A program generator may use this directive to associate errors in generated C code with the original sources. For example, the program generator **yacc** uses **#line** instructions to link the C code it generates with the **yacc** code written by the programmer.

#pragma — Preprocessing directive

Perform implementation-defined task
#pragma *preprocessing-tokens*$_{opt}$ *newline*

The preprocessing directive **#pragma** causes the implementation to behave in an implementation-defined manner. A **#pragma** might be used to give a "hint" to the translator about the best way to generate code, optimize, or diagnose errors. It may also pass information to the translator about the environment, or add debugging information. The design of **#pragma** is left up to the implementation.

Cross-references

Standard, §3.8.6
The C Programming Language, ed. 2, p. 233

See Also

preprocessing

Notes

An unrecognized pragma is ignored. Because of this subtlety, one should be careful when porting code that contains pragmas to other implementations.

As of this writing, no Mark Williams compiler uses **#pragma**.

#undef — Preprocessing directive

Undefine a macro
#undef *identifier*

The preprocessing directive **#undef** tells the C preprocessor to disregard *identifier* as a manifest constant or macro. It undoes the effect of the **#define** directive.

#undef does not give an error if *identifier* is not defined. It can also undefine macros that are predefined by the implementation, other than those specified by the Standard to be unreadable.

Cross-references

Standard, §3.8.3.5
The C Programming Language, ed. 2, p. 230

See Also

#define, preprocessing

Notes

If an implementation has defined a function both as a macro and as a library function, then the directive

```
#undef function
```

undefines the macro version, and forces the implementation to use the library version.

Some previous implementations allowed a user to "stack" macro definitions and "unstack" them by **#undef**ing them one level at a time. The Standard, however, states that one **#undef** directive undefines all previous definitions.

% — Operator

Remainder operator
operand1 **%** *operand2*

The operator **%** divides *operand1* by *operand2* and yields the remainder.

Both *operand1* and *operand2* must have integral type. Both undergo the usual arithmetic conversions before they are divided, and the type of the result is that to which the operands were converted. If *operand2* is zero, the behavior is undefined. If either operand is negative, the sign of the result is implementation-defined.

The remainder operation normally throws away the quotient. The division operator **/** returns the quotient of a division operation, and throws away the remainder. If you wish to obtain both quotient and remainder, use the functions **div** or **ldiv**. To obtain the remainder from floating-point division, use the function **fmod**.

Cross-references

Standard, §3.3.5
The C Programming Language, ed. 2, p. 205

See Also

%= — Operator

Remainder assignment operator
operand1 **%=** *operand2*

The operator **%=** divides *operand1* by *operand2* and assigns the remainder to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 % operand2
```

Each operand must have an integral type. If the value of *operand2* is zero, the result is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

%, expressions

& — Operator

&*operand*
operand1 **&** *operand2*
The operator **&** has two meanings, depending upon whether it has one operand or two. In the former instance, it yields the address of its operand. In the latter instance, it performs a bitwise AND operation upon its operands.

Address-of Operator

When used with one operand, **&** yields the value of the address of its operand in the form of a pointer to the type of its operand. The operand must be an lvalue or function designator, with the following restrictions: the operand may not be a bitfield, and it may not be declared with the storage-class specifier **register**. The resulting pointer has the type "pointer to *type*", where *type* is the type of the operand.

ANSI C allows you to take the address of a function or array.

Bitwise AND Operator

When used with two operands, **&** performs a bitwise AND operation. Each operand must have integral type. Each undergoes the normal arithmetic conversions before the operation. **&** yields a result whose type is the same as the promoted operands.

A bitwise AND operation compares the operands bit by bit. It sets a bit in the object it creates only if the corresponding bits in both operands are set.

For example, consider an environment that uses extended ASCII. Here, the character ')' has the bit pattern:

```
0010 1001
```

and the character 'L' has the bit pattern:

```
0100 1100
```

The operation **')'&'L'** yields an object with the following bit pattern:

```
0000 1000
```

Only one bit was set in the result because in only one instance were both corresponding bits set in the operands.

The **&** operation is sometimes called the "intersection" of two bit sets.

Cross-references

Standard, §3.3.3.2, §3.3.10
The C Programming Language, ed. 2, pp. 48, 93

See Also

expressions

&& — Operator

Logical AND operator
operand1 **&&** *operand2*

The operator **&&** performs a logical AND operation. Both *operand1* and *operand2* must have scalar type.

The result of this operation has type **int**. The result has a value of one if both operands are true (i.e., nonzero). If either operand is false (zero), then the result has a value of zero.

The operands are evaluated from left to right. If *operand1* is false, then *operand2* is not evaluated. If *operand2* is an expression that yields a side-effect, the results of the **&&** operation may not be what you expect. If *operand1* is false, *operand2* is not evaluated and its side-effect not generated.

Cross-references

Standard, §3.3.13
The C Programming Language, ed. 2, p. 207

See Also

||, expressions

&= — Operator

Bitwise-AND assignment operator
operand1 **&=** *operand2*

The operator **&=** performs a bitwise AND operation on *operand1* and *operand2* and assigns the result to *operand1*. It is equivalent to the expression

```
operand1 = operand1 & operand2
```

Both operands must have integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

&, expressions

() — Punctuator

functionname **(** *arguments* **)**
(*newtype* **)** *identifier*
(*primary expression* **)**
The characters **()** have two uses in the C world: as punctuators and as operators. Parentheses must be used in pairs.

When the parentheses follow an identifier, they indicate that it names a function. When used with a

function declaration, a function prototype, or a function definition, the parentheses may enclose a list of parameters for the function and the type of each parameter. When used with a function call, they enclose a list of arguments to be passed to the function.

When parentheses precede an identifier and enclose a typename alone, then they function as the cast operator. Here, the type of the identifier is changed, or *cast*, to the type enclosed within parentheses.

Finally, when parentheses enclose an expression, that expression is by definition considered to be a primary expression. This means that the expression is resolved before any outer expression is evaluated.

To see the variety of uses for **()**, consider the following expression:

```
if ((fileptr = (void *)fopen("filename", "r")) == NULL)
```

The outermost pair of parentheses enclose the arguments to **if**. The next innermost pair of parentheses enclose the expression

```
fileptr = (void *)fopen("filename", "r")
```

which must be resolved before it is compared with NULL. The pair of parentheses that enclose the type **void** * casts the object returned by **fopen** to type **void** *. Finally, the parentheses that follow **fopen** mark that identifier as a function and enclose the arguments that are passed to it, in this case the string literals **filename** and **r**.

Cross-reference

Standard, §3.1.6, §3.3.2.2, §3.3.4

See Also

function calls, function definition, function prototype, operators, punctuators

Notes

Under ANSI C, parentheses affect the grouping of expressions. This is a quiet change from the definition in the first edition of *The C Programming Language*, which allowed translators to rearrange expressions in the presence of parentheses on expressions that involved commutative and associative operators (binary **+** and *). The *as if* rule still applies in this case: if the translator can produce the same results, it is free to rearrange expressions in the face of parentheses.

* — Operator

*pointer
typename * type-qualifier-list$_{opt}$ identifier
operand1 * operand2
The character * is used both as an operator and as a punctuator.

Multiplication Operator

When the * appears between two operands with arithmetic type, it is the multiplicative operator. It multiplies its operands and yields the product. Both operands undergo normal arithmetic conversion. The type of the result is the one to which both operands were converted.

Indirection Operator

When * is used before one operand that is of a pointer type, it *dereferences* the pointer. That is, it yields the value of the object to which the pointer points. If the pointer points to a function, then the result is a function designator. If the pointer points to an object, the resulting lvalue has the type of the object to which the pointer points.

If indirection is performed on any pointer to an incomplete type, the behavior is undefined. This means that no pointer with type **void** * can be dereferenced.

Pointer Punctuator

When the * is used in a declaration, it indicates that the variable being declared is a pointer. For example, consider the following:

```
int example1;
int *example2;
```

Here, **example1** has type **int**, and **example2** has type "pointer to **int**".

Cross-references

Standard, §3.1.6, .3.3.2, §3.3.5, §3.5.4.1
The C Programming Language, ed. 2, pp. 94, 205

See Also

expressions, operators, pointer, punctuators

*/ — Comment delimiter

The characters ***/** together mark the end of a comment.

Cross-references

Standard, §3.1.9
The C Programming Language, ed. 2, p. 192

See Also

/*, comment

*= — Operator

Multiplication assignment operator
operand1 ***=*** *operand2*

The operator ***=*** multiplies *operand1* by *operand2* and assigns the product to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 * operand2
```

Each operand must have an arithmetic type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

***, expressions**

+ — Operator

+*operand*
operand1 **+** *operand2*
The operator **+** has two uses, depending upon whether it is used with two operands or one. In the former instance, it indicates that the given operand should be computed without any associative or commutative regrouping that the translator might normally apply to expressions. In the latter, it adds the two operands together.

The Unary + Operator

The unary operator **+** takes an operand that has a scalar type and yields its value. If the operand has a negative value, then a negative value is returned. The operand undergoes integral promotion,

and the type returned is that to which the operand is promoted.

The Addition Operator

The addition operator **+** adds two operands. Both operands may have arithmetic types, or one of the operands may be a pointer and the other an integral type.

If both operands have arithmetic types, then each undergoes integral conversion before addition is performed; the type of the result is the type to which both are converted.

When an integral type is added to a pointer, the value of the integral operand is first multiplied by the size of the object to which the pointer points, in bytes, and then addition is performed. The result of the addition operation returns a pointer that is appropriately offset from the pointer operand.

Pointer addition is often used for pointers that point to arrays. Note the following rules for incrementing a pointer to an array:

- If a pointer points to an array, then the result of addition will point to another member of the same array — assuming that the array is large enough.
- If a pointer to an array is incremented and the resulting pointer does *not* point to a member of the array or one past the last member, then behavior is undefined.
- Behavior is also undefined if the pointer operand and the result of the addition operation do not point to the same array object *and* the result of the addition operation is then redirected with the unary * operator. In other words, it is legal for a translator to test array bounds.

Cross-references

Standard, §3.3.3.3, §3.3.6
The C Programming Language, ed. 2, pp. 203, 205

See Also

++, -, expressions

++ — Operator

Increment operator
operand++
++*operand*

The operator **++** increments its operand. When it appears before its operand, it is called the *pre-increment operator*; when it appears after its operand, it is called the *post-increment operator*. In both cases, it is equivalent to *operand* **=** *operand***+1**. *operand* must be a modifiable lvalue.

These operators differ as follows: with the prefix operator, the value of the operand is used *after* it is incremented; whereas with the postfix operator, the value of the operand is used *before* it is incremented.

The following example illustrates the difference between the preincrement and postincrement operators.

```
#define MAX  10
int x = 0, count = 0;

/* loop 1 */
while (++x < MAX)
      count++;
```

```
/* loop 2 */
while (x++ < MAX)
      count++;
```

The first loop will iterate nine times, the second will iterate ten times. The first loop preincrements the loop variable **x** before using it within the conditional expression. The second loop, which uses the postincrement operator, first uses the current value of **x** in the conditional, then increments its value.

Cross-references

Standard, §3.3.2.4, §3.3.3.1
The C Programming Language, ed. 2, p. 46

See Also

--, expressions

+= — Operator

Addition assignment operator
operand1 **+=** *operand2*

The operator **+=** adds the value of *operand1* with that of *operand2* and stores the sum within *operand1*. It is equivalent to the expression:

```
operand1 = operand1 + operand2
```

Both operands have arithmetic types, or *operand1* has a pointer type and *operand2* has integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

-=, expressions

Notes

The lvalue *operand1* is evaluated only once.

, — Operator

identifier1 **,** *identifier2*
expression1 **,** *expression2*
The character ',' can be used as punctuator or an operator.

The Comma Punctuator

When it is used as a punctuator, the comma separates the parameters in a function declaration, the parameters to a function-like macro, the arguments to a function call, or the items in a list of identifiers. For example, in the expression

```
int foo, bar, baz;
```

the comma separates the identifiers being declared, all of which are of type **int**.

The Comma Operator

When used outside of a declaration or parameter list, the comma acts as an operator. The comma operator evaluates its left argument first, then its right argument. The value and type of the comma expression is that of the right operand.

LEXICON

For example, the following shows how the comma operator is used in a loop:

```
int i, j;
   . . .
for (i=j=0; i<10 && j<25; i++, j++);
```

This loop uses the comma operator to help increment two variables upon each iteration.

Cross-references

Standard, §3.3.17
The C Programming Language, ed. 2, p. 62

See Also

expressions

Notes

A comma expression cannot be an lvalue.

- — Operator

-operand
operand1 - operand2
The operator **-** has two uses, depending upon whether it is used with two operands or one. In the former situation, it subtracts the operand to its right from the operand to its left. In the latter, it returns the negated value of its operand.

Subtraction Operator

The operator **-** can subtract the following operands from each other:

- Two arithmetic types.
- Two pointers to objects that have compatible types and compatible qualification.
- Two pointers that point to objects that have compatible types, but not necessarily compatible qualification.
- An integral type from a pointer.

When both operands have arithmetic type, each undergoes integral promotion. The type of the result is that to which the operands were promoted. Its value is the difference when the right operand is subtracted from the left.

When one pointer is subtracted from another, the result is of type **ptrdiff_t**. This type is defined in the header **stddef.h**. If two pointers that do not point to the same array are subtracted from each other, behavior is undefined. The only exception is the expression

```
(X+1) - X
```

where, if **X** points to the last member of the array, the result is one by definition.

If two pointers that point to the same array are subtracted from each other, the result is automatically divided by the size of an array member. This yields a value that is the same as would result if the two appropriate array subscripts had been subtracted from each other. If the result of pointer subtraction points past the end of an array, the behavior is undefined. The sole exception, again, is the expression given above.

When subtracting a scalar from a pointer, the result is as if the scalar were multiplied by the size of the object pointed to by the pointer, and then subtracted.

The unary operator **-** takes an operand with arithmetic type. The operand first undergoes normal integral promotion. The type of the resulting expression is the one to which the operand was promoted; and the value of the resulting expression is the negated value of the operand.

Cross-references

Standard, §3.3.3.3, §3.3.6
The C Programming Language, ed. 2, pp. 203, 205

See Also

+, --, expressions

-- — Operator

Decrement operator
*operand***--**
--*operand*

The operator **--** decrements its operand. When it appears before its operand, it is called the *pre-decrement operator*; when it appears after its operand, it is called the *post-decrement operator*. In both cases, it is equivalent to *operand* **=** *operand* **- 1**.

These operators differ as follows: with the prefix operator, the value of the operand is used *after* it is decremented; whereas with the postfix operator, the value of the operand is used *before* it is decremented.

Cross-references

Standard, §3.3.2.4, §3.3.3.1
The C Programming Language, ed. 2, p. 46

See Also

++, expressions

-= — Operator

Subtraction assignment operator
operand1 **-=** *operand2*

The operator **-=** subtracts the value of *operand2* from that of *operand1* and stores the difference within *operand1*. It is equivalent to the expression:

```
operand1 = operand1 - operand2
```

Both operands have arithmetic types, or *operand1* has pointer type and *operand2* has integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

+=, expressions

-> — Operator

Select a member
objectpointer **->** *membername*

The operator **->** selects a member of a structure or a **union** through a pointer.

objectpointer must point to a structure or **union**. *membername* must name a member of the structure or **union** to which *objectpointer* points. For example, consider the following:

```
struct example {
      int member1;
      long member2;
      example *member3;
};

struct example structure;
struct example *pointer = &structure;
```

To select **member1** within **structure** via **pointer**, use the expression:

```
pointer->member1
```

Behavior is implementation-defined if one member of a **union** is accessed after another member has been stored within the **union**.

Cross-references

Standard, §3.3.2.3
The C Programming Language, ed. 2, p. 131

See Also

., expressions, operators

. — Operator

Member selection
objectname **.** *membername*

The operator **.** is used to select a member of a structure or a **union**.

objectname must name a structure or **union**. *membername* must be a member of the structure or **union** that *objectname* names. For example, consider the following:

```
struct example {
      int member1;
      long member2;
      example *member3;
};

struct example object;
```

To read **member1** within **object**, use the expression:

```
object.member1
```

Cross-references

Standard, §3.3.2.3
The C Programming Language, ed. 2, p. 128

See Also

->, expressions, member

/ — Operator

Division operator
operand1 **/** *operand2*

The operator **/** divides *operand2* by *operand1* and yields the quotient. Each operand must have arithmetic type and undergoes the usual arithmetic promotion before the operation is performed. The result of the operation has the type to which the operands are promoted. If the result of **X/Y** can be represented, then **(X/Y)*Y+(X%Y)** must equal **X**.

If *operand2* is zero, the result is undefined. If either operand is negative, the result is either the largest integer that is less than the algebraic quotient, or the smallest integer that is greater than the algebraic quotient, whichever the implementation prefers. For example, in the expression

```
7 / -2
```

the algebraic quotient is **-3.5**. The implementation determines whether the result is **-4** (the largest integer less than the algebraic quotient) or **-3** (the smallest integer greater than the algebraic quotient).

The division operation normally throws away the remainder. The remainder operator **%** returns the remainder of a division operation and throws away the quotient. If you wish to obtain both quotient and remainder, use the functions **div** or **ldiv**.

Cross-references

Standard, §3.3.5
The C Programming Language, ed. 2, p. 205

See Also

%, div, expressions, ldiv

/* — Comment delimiter

The characters **/*** together mark the beginning of a comment.

Cross-references

Standard, §3.1.9
The C Programming Language, ed. 2, p. 192

See Also

***/, comment**

/= — Operator

Division assignment operator
operand1 **/=** *operand2*

The operator **/=** divides *operand1* by *operand2*, and assigns the quotient to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 / operand2
```

Each operand must have arithmetic type.

If the value of *operand2* is zero, behavior is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

/, expressions

: — Punctuator

When punctuator **:** follows an identifier, it marks the identifier as being a label. When it precedes an integer constant in the declaration of a structure or **union**, it marks the constant as giving the size of a bit-field.

Cross-reference

Standard, §3.1.6
The C Programming Language, ed. 2, p. 66

See Also

?:, bit-fields, goto, label, punctuators

; — Punctuator

The punctuator **;** marks the end of a statement.

Cross-reference

Standard, §3.1.6

See Also

punctuators, statements

< — Operator

Less-than operator
operand1 **<** *operand2*

The operator **<** compares two operands. It yields one if *operand1* is less than *operand2*, and zero if *operand1* is greater than or equal to *operand2*.

See **operators** for more information on the types of operands that can be compared.

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<=, >, expressions

<< — Operator

Bitwise left-shift operator
operand1 **<<** *operand2*

The operator **<<** shifts the bits in *operand1* to the left by *operand2* places. This is called the *bitwise left shift* operation.

Both operands must have integral types. Both undergo the usual arithmetic conversions, and the result has the type to which the left operand was promoted.

A bitwise left-shift operation moves the bits of an object to the left, and fills the vacated bits with zeroes. For example, consider an environment that uses extended ASCII. Here, the character constant **'?'** has the bit pattern:

```
0011 1111
```

In this environment, the expression

```
'?' << 4
```

yields the following pattern of bits:

```
0000 0011 1111 0000
```

The "nybbles" to the left result from the promotion of the **char** to type **int**. All bits are shifted four places to the left, and the four vacated bits to the right are filled with zeroes.

The left-shift operation is sometimes called the "logical" shift operation, which will fill vacated bits with zeroes.

If *operand2* is negative or is larger than the number of bits in *operand1*, behavior is undefined.

Example

For a practical example of the operator **<<**, see **rand()**.

Cross-references

Standard, §3.3.7
The C Programming Language, ed. 2, pp. 48, 207

See Also

<<=, >>, expressions

<<= — Operator

Bitwise left-shift assignment operator
operand1 **<<=** *operand2*

The operator **<<=** shifts the bits in *operand1* to the left by *operand2* places, and assigns the result to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 << operand2
```

Both operands must have integral type.

If *operand2* is negative or has a value greater than the number of bits in *operand1*, behavior is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

<<, expressions

<= — Operator

Less-than or equal-to operator
operand1 **<=** *operand2*

The operator **<=** compares two operands. It returns one if *operand1* is less than or equal to *operand2*, and it returns zero if *operand1* is greater than *operand2*.

See **operators** for more information on the types of operands that can be compared.

Example

For an example of using this operator in a program, see **bitwise operators**.

LEXICON

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<, >=, expressions

= — Operator

Assignment operator
operand1 **=** *operand2*

The operator **=** copies the value of *operand2* into *operand1*. The value of *operand2* is converted to the type of *operand1* before they are copied.

The following types of operands are allowed:

- Both have an arithmetic type. *operand1* may be qualified.
- Both are compatible structures or **union**s. *operand1* may be qualified.
- Both are pointers to compatible types. *operand1* may be a pointer to a qualified type. *operand2* may be NULL. Either may be of type **void** *, assuming the other points to an object or an incomplete type.

operand1 must be a modifiable lvalue.

Cross-references

Standard, §3.3.16.1
The C Programming Language, ed. 2, pp. 50, 208

See Also

==, expressions

== — Operator

Equality operator
operand1 **==** *operand2*

The operator **==** compares *operand1* with *operand2*. The result is one if the operands are equal, and zero if they are not.

The operands must be one of the following:

- Arithmetic types.
- Pointers to compatible types (ignoring qualifiers on these types).
- A pointer to an object or incomplete type and a pointer to **void**.
- A pointer and NULL.

If both operands have arithmetic type, they undergo usual arithmetic conversion before being compared. If one operand is a pointer to an object and the other is a pointer to **void**, the pointer to an object is converted to a pointer to **void** for purposes of the comparison.

If two pointers to functions compare equal, then they point to the same function; likewise, if two pointers to data objects compare equal, then they point to the same object. However, on machines that provide separate spaces for instructions and data, a pointer to a function may compare equal to a pointer to a data object. Therefore, you should not depend on being able to distinguish function pointers from data object pointers by value. Further, on machines that allow many pointer values

to refer to the same object (e.g., i8086 LARGE model), two pointers that do not compare equal may nonetheless point to the same object.

Cross-references

Standard, §3.3.9
The C Programming Language, ed. 2, pp. 41, 207

See Also

!=, expressions

Notes

Perhaps the commonest mistake made by C programmers is to use the assignment operator '=' in place of the equality operator '==' where a conditional expression is expected. For example:

```
if (variable1 = variable2)    /* WRONG */
      dosomething();
```

Here, the value of **variable2** is copied into **variable1**; whether the expression succeeds or not depends upon the value of **variable2** rather than the equality of the two variables. Hence, the condition will be true as long as this operand has a value other than zero. This code will translate, often without generating a warning message, but probably will not run correctly.

Type conversion will affect comparison, particularly if a **char** is being compared with an integral type with a negative value. For example, consider the comparison:

```
char variable;
   . . .
if (variable == -1)
      dosomething();
```

Here, **variable** is promoted to an **int** before it is compared with **-1**. However, if **char** is unsigned by default, when it is expanded, it can never compare equal to a negative number. For maximum portability, when using **char**s that may take negative values, declare them as type **int** or type **signed char**. All Mark Williams compilers used signed **char**s by default.

Comparing **float**s and **double**s for equality is usually a mistake, especially as a control expression in a loop. Implementations of floating-point arithmetic are often inexact.

> — Operator

Greater-than operator
operand1 **>** *operand2*

The operator **>** compares two operands. It returns one if *operand1* is greater than *operand2*. It returns zero if *operand1* is less than, or equal to, *operand2*.

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<, >=, expressions

>= — Operator

Greater-than or equal-to operator
operand1 **>=** *operand2*

LEXICON

The operator **>=** compares two operands. It returns one if *operand1* is greater than, or equal to, *operand2*; it returns zero if *operand1* is less than *operand2*.

Cross-references

Standard, §3.3.8
The C Programming Language, ed. 2, pp. 41, 206

See Also

<=, >, operators

>> — Operator

Bitwise right-shift operator
operand1 **>>** *operand2*

The operator **>>** shifts the bits in *operand1* to the right by *operand2* places. This is called the *bitwise right shift* operation.

Both operands must have integral type. Both undergo the usual arithmetic conversions, and the result has the type to which the left operand was promoted.

A bitwise right-shift operation moves the bits of an object to the right. The vacated bits are filled with zeroes, unless *operand1* is signed and has a negative value. In that case, the vacated bits will propagate the sign bit (i.e., be filled with ones).

For example, consider an environment that uses extended ASCII. Here, the character constant **'?'** has the bit pattern:

```
0011 1111
```

In this environment, the expression

```
'?' >> 4
```

yields the following pattern of bits:

```
0000 0000 0000 0011
```

The two "nybbles" to the right result from the promotion of the **char** to type **int**. All bits are shifted four places to the right, and the four vacated bits to the left are filled with zeroes. The nybble **1111** disappears.

The right-shift operation is sometimes called the "arithmetic" shift operation.

If *operand2* is negative or is larger than the number of bits in *operand1*, behavior is undefined.

Example

For an example of using this operator in a program, see **srand**.

Cross-references

Standard, §3.3.7
The C Programming Language, ed. 2, pp. 48, 207

See Also

<<, >>=, expressions

>>= — Operator

Bitwise right-shift assignment operator
operand1 **>>=** *operand2*

The operator **>>=** shifts the bits in *operand1* to the right by *operand2* places, and assigns the result to *operand1*. It is equivalent to the expression:

```
operand1 = operand1 >> operand2
```

Both operands must have integral type.

If *operand2* is negative or has a value larger than the number of bits in *operand1*, behavior is undefined.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

>>, expressions

?: — Operator

Conditional operator
conditional **?** *expression1* **:** *expression2*

The conditional operator **?:** causes one or the other of two expressions to be executed.

If the *conditional* evaluates to true (nonzero), then *expression1* is evaluated; otherwise, *expression2* is evaluated. The operator as a whole yields the result of whichever expression is executed.

The *logical-OR-expression* must have a scalar type. The conditional operator may take the following types:

- Both are arithmetic types. Each undergoes normal arithmetic conversion, and the result has the type to which they are converted.
- Both have compatible structure or **union** types. They are converted to a common type, and the result has that type.
- Both are **void** types. The result is of type **void**.
- Both are pointers to compatible types, whether qualified or unqualified. The result is a pointer that is qualified by all the qualifiers of both operands.
- One is a pointer and the other NULL. The result is of the pointer's type.
- One points to an object or incomplete type, and the other is type **void** *. Both operands are converted to type **void** * before evaluation, and the result also has that type.

The logical expression can also be a scalar identifier, constant, or function.

Cross-references

Standard, §3.3.15
The C Programming Language, ed. 2, p. 51

See Also

expressions

LEXICON

Notes

The conditional operator does not yield an lvalue. For example:

```
int x, a, b;
(x ? a : b) = 5;  /* WRONG */
```

is incorrect, but

```
int x;
int *ptr1, *ptr2;
*(x ? ptr1 : ptr2) = 5; /* RIGHT */
```

is correct.

[] — Operator

Array subscript operator
*arrayname***[***size***]**

The array-subscript operator **[]** is used in different contexts. It is used to declare an array, with or without the array size. It is used as a subscript operator, and it can also be used when passing an array as an argument. *arrayname* is the name of the array to be accessed; *size* is the number of objects in the array.

The Standard states that one of the items *arrayname* or *size* must be a pointer and the other an integer. To calculate the address of an element within an array, the integer is multiplied by the size of an element of the array, and the product added to value of the pointer. In most C programs, *arrayname* gives the pointer and *size* the integer offset.

The operator **[]** can also be used to select an object within an array; the objects are numbered from zero through *size*-1. For example, if *arrayname* points to an array of **int**s, and if *size* is equal to six, then the expression

```
arrayname[4]
```

is equivalent to:

```
*(arrayname+4)
```

This expressions yields not an address, but the contents of the array at the requested point.

An array can be followed by more than one pair of brackets. Such arrays are called *multidimensional*. To see how such an array works, consider the following multidimensional array:

```
#define DIMENSION1 5
#define DIMENSION2 10
int arrayname[DIMENSION1][DIMENSION2];
```

Here, **dimension1** holds five objects, each of which is the size set by **dimension2**: in this instance, ten **int**s. Thus, the expression

```
arrayname[3][5];
```

is equivalent to writing:

```
*(arrayname+(3*DIMENSION2)+5)
```

An expression of the form

```
arrayname[3];
```

indicates an entire row of the array. This is sometimes called a “slice”.

Cross-references

Standard, §3.3.2.1
The C Programming Language, ed. 2, pp. 97*ff*

See Also

array, expressions

Notes

Given the Standard's description of how an array is accessed, the elements of an array access may be reversed. For example, given the following code,

```
int arrayname[5];
int counter = 3;
```

the expressions

```
arrayname[counter]
```

and

```
counter[arrayname]
```

should yield the same result. Using these expressions interchangeably will result in programs that are very hard to read and maintain.

^ — Operator

Bitwise exclusive OR operator
operand1 **^** *operand2*

The operator **^** performs an bitwise exclusive OR operation.

Each operand must have integral type, and each undergoes the usual arithmetic conversions. The result has integral type.

A bitwise exclusive OR operation compares the bit patterns of the operands, then sets each bit in its result if either, but not both, of the corresponding bits in the operands is set.

For example, consider an environment which uses extended ASCII. In this environment, the character **9** is represented by the bit pattern

```
0011 1001
```

and the character **w** by the bit pattern:

```
0111 0111
```

Thus, the operation:

```
'9' ^ 'w';
```

yields the following bit pattern:

```
0000 0000 0100 1110
```

The extra "nybbles" to the left are created by the promotion of the character constants to type **int**. If the corresponding bits in the operands were both set to one, the bit in the result was set to zero.

Example

For an example of using this operator in a program, see **srand**.

Cross-references

Standard, §3.3.11
The C Programming Language, ed. 2, pp. 48, 207

See Also

^=, |, expressions

^= — Operator

Bitwise exclusive-OR assignment operator
operand1 **^=** *operand2*

The operator **^=** performs a bitwise exclusive-OR operation on *operand1* and *operand2*, and assigns the result to *operand1*. It is equivalent to the expression

```
operand1 = operand1 ^ operand2;
```

Both operands must have integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

^, expressions

_ _DATE_ _ — Manifest constant

Date of translation

_ _**DATE**_ _ is a manifest constant that is defined by the implementation. It represents the date that the source file was translated. It is a string literal of the form

```
"Mmm dd yyyy"
```

where **Mmm** is the same three-letter abbreviation for the month as is used by **asctime**; **dd** is the day of the month, with the first **d** being a space if translation occurs on the first through the ninth day of the month; and **yyyy** is the current year. If the date of translation is not available, then a valid, implementation-defined date must be supplied.

The value of _ _**DATE**_ _ remains constant throughout the processing of the translation unit. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _FILE_ _, _ _LINE_ _, _ _STDC_ _, _ _TIME_ _, preprocessing

__end — External data

extern char * __end;
__end is an external variable that points to the end of your program's data space. It is set by the C runtime startup, and can be incremented by the function **sbrk**.

See Also

Environment, malloc, maxmem, sbrk

_ _FILE_ _ — Manifest constant

Source file name

_ _FILE_ _ is a manifest constant that is defined by the implementation. It represents, as a string constant, the name of the current source file being translated.

_ _FILE_ _ may not be the subject of a **#define** or **#undef** preprocessing directive, but it may be altered with the **#line** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

#line, _ _DATE_ _, _ _LINE_ _, _ _STDC_ _, _ _TIME_ _, preprocessing

_ _LINE_ _ — Manifest constant

Current line within a source file

_ _LINE_ _ is a manifest constant that is defined by the implementation. It represents the current line within the source file. The Standard defines the current line as being the number of newline characters read, plus one.

_ _LINE_ _ may not be the subject of a **#define** or **#undef** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _DATE_ _, _ _FILE_ _, _ _STDC_ _, _ _TIME_ _, preprocessing

_ _STDC_ _ — Manifest constant

Mark a conforming translator

_ _STDC_ _ is a manifest constant that is defined by the implementation. If it is defined to be equal to one, then it indicates that the translator conforms to the Standard.

The value of **_ _STDC_ _** remains constant throughout the entire program, no matter how many source files it comprises. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Example

For an example of using **_ _STDC_ _** in a program, see **assert**.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _TIME_ _, preprocessing

Notes

If an implementation is not fully compatible with the Standard, then it should not define **_ _STDC_ _**. A value greater than one may indicate compliance with a later version of the Standard.

LEXICON

_ _TIME_ _ — Manifest constant

Time source file is translated

_ _TIME_ _ is a manifest constant that is defined by **Let's C**. It represents the time that a source file is translated. It is a string literal of the form:

```
"hh:mm:ss"
```

This is the same format used by the function **asctime**. If the time of translation is not available, then a valid, implementation-defined string must be supplied.

The value of this remains constant throughout the processing of the translation unit. It may not be the subject of a **#define** or **#undef** preprocessing directive.

Cross-references

Standard, §3.8.8
The C Programming Language, ed. 2, p. 233

See Also

_ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _STDC_ _, preprocessing

_exit() — Extended function (libc)

Terminate a program
int _exit(int *status***);**

_exit terminates a program directly. It returns *status* to the calling program, and exits.

Unlike the library function **exit**, **_exit** does not perform extra termination cleanup, such as flushing buffered files and closing open files.

_exit should be used only in situations where you do *not* want buffers flushed or files closed, such as when your program detects an irreparable error condition and you want to "bail out" to keep your data files from being corrupted.

_exit should also be used with programs that do not use STDIO and have been compiled with the **-ns** option to the **cc** command. Unlike **exit**, **_exit** does not use STDIO. This will help you create programs that are extremely small when compiled.

See Also

exit, extended miscellaneous, runtime startup, system

_tolower() — Extended macro (xctype.h)

Convert letter to lower case
#include <xctype.h>
int _tolower(int *c***);**

The macro **_tolower** converts *c* to lower case and returns it. If *c* is not a letter, the result is undefined.

_tolower differs from its cousin **tolower** in that **_tolower** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **tolower** is a function that does check its argument.

Example

This example opens a file of text and reverses the cases of all characters. It demonstrates **_tolower** and **_toupper**.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <xctype.h>

void
fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *fp;
      int ch;

      if (--argc != 1)
            fatal("Usage: example filename");

      if ((fp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open file for reading");

      while ((ch = fgetc(fp)) != EOF) {
            if ((isascii(ch) != 0) && ch != '\r')
                  fatal("Not a text file");

            if (isalpha(ch) != 0)
                  fputc((isupper(ch) ? _tolower(ch) : _toupper(ch)),
                        stdout);
            else
                  fputc(ch, stdout);
      }
      return EXIT_SUCCESS;
}
```

See Also

_toupper, character handling, tolower

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **xctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

_toupper() — Extended macro (xctype.h)

Convert letter to upper case
#include <xctype.h>
_toupper(int *c***);**

The macro **_toupper** returns *c* converted to upper case. If *c* is not a letter, the result is undefined.

_toupper differs from its cousin **toupper** in that **_toupper** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **toupper** is a function that does check its argument.

Example

For an example of this routine, see the entry for **_tolower**.

See Also

_tolower, character handling, toupper

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **xctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

_zero() — i8086 support (libc)

Zero a block of memory
void _zero(unsigned *offs***, unsigned** *seg***, unsigned** *n***);**

_zero zeros out *n* bytes of memory at the address given by the segment *seg* and the offset *offs*.

_zero requires the full offset/segment address to work properly. If your program is compiled into SMALL model, you should use the macro **PTR** to ensure that a full address is used.

Example

The following example initializes a chunk of memory, displays it, and then zeroes it out.

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      char foo[80] = "Here is a string.";
      printf("Before _zero: %s\n", foo);

      _zero(PTR(foo), 80);
      printf("\nAfter _zero: %s\n", foo);
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, PTR

{} — Punctuator

The punctuators {}, or “braces”, are used to delimit a block, and to group initializers. Braces must be used in pairs.

Cross-reference

Standard, §3.1.6

See Also

block, initialization, punctuators

| — Operator

Bitwise inclusive OR operator
operand1 **|** *operand2*

The operator **|** performs an bitwise inclusive OR operation. Each operand must have integral type. Each undergoes the usual arithmetic conversions, and the result has integral type.

A bitwise inclusive OR operation compares the bit patterns of the operands. It then sets each bit in the result if either, or both, of the corresponding bits in each of the operands is set.

For example, consider an environment which uses extended ASCII. Here, the character **9** is represented by the bit pattern

```
0011 1001
```

and the character **w** by the bit pattern:

```
0111 0111
```

Thus, the operation:

```
'9' | 'w'
```

yields the following bit pattern:

```
0000 0000 0111 1111
```

The extra "nybbles" to the left are created by the promotion of the character constants to type **int**.

The bitwise inclusive OR operation is also called the "union" of two bitsets.

Cross-references

Standard, §3.3.12
The C Programming Language, ed. 2, pp. 48, 207

See Also

^, |=, expressions

|= — Operator

Bitwise inclusive-OR assignment operator
operand1 **|=** *operand2*

The operator **|=** performs a bitwise inclusive OR operation on *operand1* and *operand2*, and assigns the result to *operand1*. It is equivalent to the expression

```
operand1 = operand1 | operand2
```

Both operands must have integral type.

Cross-references

Standard, §3.3.16.2
The C Programming Language, ed. 2, pp. 50, 208

See Also

|, expressions

|| — Operator

Logical OR operator
operand1 **||** *operand2*

The operator **||** performs a logical OR operation. Both *operand1* and *operand2* must have scalar type.

The result of the **||** operation has type **int**. The value of the result is one if either operand is true (nonzero); if both operands are false (equal to zero), the result has a value of zero.

The operands are evaluated from left to right. If *operand1* is true, then *operand2* is not evaluated. If *operand2* is an expression that yields a side-effect, the results of the **||** operation may not be what you expect: if *operand1* is true, *operand2* is not evaluated and its side-effect not generated.

LEXICON

Cross-references

Standard, §3.3.14
The C Programming Language, ed. 2, p. 208

See Also

&&, expressions

~ — Operator

Bitwise complement operator
~*operand*

The operator **~** is the bitwise complement operator. Its operand has an integral type, which undergoes integral promotion. The result is an object whose type is that of the promoted operand and whose bit pattern inverts that of the operand. This is also called a "one's complement operation".

For example, consider the object:

```
char example = 'a';
```

In an environment that uses extended ASCII, **example** will have the following bit pattern:

```
0110 0001
```

Thus, the expression **~example** promotes **example** to an **int**, and then generates an object with the following bit pattern:

```
1111 1111 1001 1110
```

As can be seen, the lower eight bits have been flipped. The eight bits on the left were added when the object was promoted to **int**. These new bits were initially set to zeroes when the character was promoted to an **int**, then the complement operation flipped the zeroes to ones. In this case, the sign bit is said to *propagate*.

Cross-references

Standard, §3.3.3.3
The C Programming Language, ed. 2, p. 204

See Also

!, expressions, integral promotion