# ! to ~

## # — Preprocessing Operator

String-ize operator

The preprocessing operator **#** can be used within the replacement list of a function-like macro. It and its operand are replaced by a string literal, which names the sequence of preprocessing tokens that replaces the operand throughout the macro.

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)");
```

Here, the preprocessor replaced **#x** with a string literal that gives 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 are deleted.
3. The original spelling of the preprocessing tokens is preserved. This means that you must take care to preserve certain characters: a backslash '\' should be inserted before every quotation mark "" that marks a string literal, and before every backslash that introduces a character constant.

### *Example*

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

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

void show(value, name)
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()
{
      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)));
      }
}
```

### See Also

**## #define, C preprocessor**
ANSI Standard, §6.8.3.2

## ## — Preprocessing Operator

Token-pasting operator

The preprocessing operator **##** 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, **##** 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. This gives you an easy way to print any one of a set of strings.

**##** 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.

For more information on object-like and function-like macros, see **#define**.

### See Also

**# #define, C preprocessor**
ANSI Standard, §6.8.3.3

### Notes

Some C implementations allow token pasting by using an empty comment. For example:

```
variable/**/number
```

The COHERENT C compiler does not recognize this "trick" because it is not consistent with the Kernighan & Ritchie standard for C, which states that a comment is white space and therefore is a token separator. In any event, token pasting should always be performed with **##**.

The **##** operator may be used only within the replacement text of a preprocessor macro definition.

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

## #define — Preprocessing Directive

Define an identifier as a macro

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

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

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, 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);
```

A given *identifier* is replaced only once by a given *replacement-list*. This is to prevent such code as

```
#define FOO FOO
```

or

```
#define FOO BAR
#define BAR FOO
```

from generating an infinite loop.

A function-like macro is more complex. It has the syntax:

`#define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list*

The preprocessor looks for *identifier*, which is a macro that resembles a function in that it is followed by a pair of parentheses that may enclose an *identifier-list*. It replaces *identifier* with the contents of *replacement-list*, up to the first lparen '(' within *replacement-list*.

The preprocessor then examines *identifier-list* for further macros, which it expands. The modified *identifier-list* is then replaced with the rest of *replacement-list*. Pairs of parentheses that are nested between the lparen that begins *replacement-list* and the ')' that ends it are copied into *identifier-list* as literal characters. The identifiers within *identifier-list* are preserved after it has been modified by *replacement-list*. The only exceptions are identifiers that are prefixed by the preprocessing operators **#** or **##**; these are handled appropriately.

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.

## Example

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

## See Also

**#, ##, #undef, C preprocessor**
ANSI Standard, §6.8.3

## Notes

A macro expansion always occupies exactly one line, no matter how many lines are spanned by the definition or the actual parameters. If you have defined macros that span more than one line, you must either redefine them to occupy one line, or somehow embed the newline character within the macro itself; otherwise, the macro will not expand correctly.

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.

Some implementations allowed a user to re-define a macro with a new **#define** directive. The Standard, however, allows only a "benign" redefinition; that is, the body of the new definition must exactly match the old definition, including parameter names and white space.

## *#elif* — Preprocessing Directive

Include code conditionally

The preprocessing directive **#elif** conditionally includes code within a program. It can be used after any of the instructions **#if**, **#ifdef**, or **#ifndef**.

If the conditional expression of the preceding **#if**, **#ifdef**, or **#ifndef** directive is false (i.e., evalutates to zero) and if the current condition is true (i.e., evaluates to a value other than 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.

### *See Also*

**#else, #endif, #if, #ifdef, #ifndef, C preprocessor, defined**
ANSI Standard, §6.8.1

## *#else* — Preprocessing Directive

Include code conditionally

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 the conditional expressions of all preceding directives evaluate to false (i.e., to zero), then the code introduced by **#else** is included within the program, up to the **#endif** directive.

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

### *See Also*

**#elif, #endif, #if, #ifdef, #ifndef, C preprocessor**
ANSI Standard, §6.8.1

## *#endif* — Preprocessing Directive

End conditional inclusion of code

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**.

### *See Also*

**#elif, #else, #if, #ifdef, #ifndef, C preprocessor**
ANSI Standard, §6.8.1

## *#if* — Preprocessing Directive

Include code conditionally

The preprocessing directive **#if** tells the preprocessor that if *constant-expression* is true (i.e., that it evalutes to a value other than zero), 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.

If *constant-expression* is an undefined symbol, the preprocessor treats it the same as it would a false statement.

### See Also

**#elif, #else, #endif, #ifdef, #ifndef, C preprocessor, defined**
ANSI Standard, §6.8.1

## *#ifdef* — Preprocessing Directive

Include code conditionally

The preprocessing directive **#ifdef** checks whether *identifier* has been defined as a macro name. If *identifier* has been defined as a macro, 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**.

### See Also

**#elif, #else, #endif, #if, #ifndef, C preprocessor, defined**
ANSI Standard, §6.8.1

## *#ifndef* — Preprocessing Directive

Include code conditionally

The preprocessing directive **#ifndef** checks whether *identifier* has been defined as a macro name. If *identifier* has *not* been defined as a macro, 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.

### See Also

**#elif, #else, #endif, #if, #ifndef, C preprocessor, defined**
ANSI Standard, §6.8.1

## *#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 (**<***header***.h>**), or it is enclosed within quotation marks (**"***header***.h"**). Angle brackets tell **cpp** to look for *file***.h** in the directories named with the **-I** options to the **cc** command line, and then in the standard directory. 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 standard directory.

Most often, the file being included is a *header*, which is a file that contains function prototypes, macro definitions, and other useful material; as its name implies, it most often appears at the head of a program. The header name must be a string of characters, possibly followed by a period '.' and a single letter, usually (but not always) 'h'. A header name may have up to 12 characters to the left of the period, and names may be case sensitive.

**#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.

Note, too, that a subordinate header file is sought relative to the original source file, rather than relative to the header that calls it directly. For example, suppose that 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 COHERENT looks for **header2.h** not in directory **/usr**, but in directory **/v/fred**.

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.

### See Also

**header files, C preprocessor**
ANSI Standard §6.8.2

### Notes

If the header's name is enclosed within quotation marks note that the name is *not* a string literal, although it looks exactly like one. Thus, a backslash '\' does not introduce an escape character.

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

The mapping provided for included files may map a given name either to an actual file, or to a member in a partitioned data set.

## *#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 ANSI 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 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.

### See Also

**C preprocessor**
ANSI Standard, §6.8.4

### 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-specific preprocessing

**#pragma** is the C preprocessing directive that triggers implementation-specific behavior. The ANSI Standard demands that every conforming implementation of C document what **#pragma** does.

COHERENT recognizes one use of **#pragma**:

```
#pragma align [n]
```

This directive permits COHERENT to conform to the Intel Binary Compatability Standard (BCS), which specifies alignment requirements for **struct**s.

The BCS requires that a **struct** be aligned consistently with the alignment of its most strictly aligned member. For example, the structure

```
struct s {
      short s_s1;
      int   s_i;
      short s_s2;
};
```

must put member **s_i** at offset 4, not 2 (because **int** is dword-aligned). If you have an array of **struct s** objects, the second will be at offset 12, not 10 (or 8), because **struct s** itself must also be dword-aligned.

This, unfortunately, creates problems with existing compiled code, and with some standards, e.g., COFF. For example, a **struct filsys** (a COHERENT file system, e.g., on a floppy or hard disk) is defined in **<sys/filsys.h>** as starting out just like the above:

```
struct filsys {
      unsigned short    s_isize;
      daddr_t           s_fsize;
      short      s_nfree;
      ...
};
```

Because **daddr_t** is **long**, COHERENT would compile this and expect to find **s_fsize** at offset 4 (not 2) and **s_nfree** at offset 8 (not 6); but this is not where the bits actually fall on an existing file system. So we circumvent the BCS with **#pragma align**. The directive **#pragma align** *n* means "align objects on *n*-byte boundaries, at most," and **#pragma align** means "restore default alignment." Thus, **<sys/filsys.h>** is edited to read:

```
struct filsys {
      unsigned short    s_isize;
#pragma align 2
      daddr_t           s_fsize;
#pragma align
      short      s_nfree;
      ...
};
```

and the compiler thinks the struct members fall at offsets 0, 2 and 6, which preserves compatibility with existing binary objects.

### See Also

**cpp, C preprocessor**
ANSI Standard, §6.8.6

## *#undef* — Preprocessing Directive

Undefine a macro
**#undef** *identifier*

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

### See Also

**#define, C preprocessor**
ANSI Standard, §6.8.3

## *__DATE__* — Manifest Constant

Date of translation

**__DATE__** is a preprocessor constant that is defined by the C preprocessor. 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.

The value of **__DATE__** remains constant throughout the processing of the a module of source code. It may not be the subject of a **#define** or **#undef** preprocessing directive.

### Example

The following prints the preprocessor constants set by the ANSI standard.

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

```
main(void)
{
      printf("Date: %s\n", __DATE__);
      printf("Time: %s\n", __TIME__);
      printf("File: %s\n", __FILE__);
      printf("Line No.: %d\n", __LINE__);

      printf("ANSI C? ");
#ifndef __STDC__
      printf("no0);
#else
      printf("ANSI C? %s(%d)0, __STDC__ ? "Yes" : "No", __STDC__);
#endif /* _defined(__STDC__) */

      exit(EXIT_SUCCESS);
}
```

### See Also

**__FILE__, __LINE__, __STDC__, __TIME__, manifest constant**
ANSI Standard, §6.8.8

## __FILE__ — Manifest Constant

Source file name

**__FILE__** is a preprocessor constant that is defined by the C preprocessor. 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.

### Example

For an example of how to use **__FILE__** in a program, see **__DATE__**.

### See Also

**__DATE__, __LINE__, __STDC__, __TIME__, manifest constant**
ANSI Standard, §6.8.8

## __LINE__ — Manifest Constant

Current line within a source file

**__LINE__** is a preprocessor constant that is defined by the C preprocessor. It represents the current line within the source file. The ANSI 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.

### Example

For an example of how to use **__LINE__** in a program, see **__DATE__**.

### See Also

**__DATE__, __FILE__, __STDC__, __TIME__, manifest constant**
ANSI Standard, §6.8.8

## __STDC__ — Manifest Constant

Mark a conforming translator

**__STDC__** is a preprocessor constant that is defined by the C preprocessor. If it is defined to be equal to one, then it indicates that the translator conforms to the ANSI 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 **__DATE__**.

### See Also

**__DATE__, __FILE__, __LINE__, __TIME__, manifest constant**
ANSI Standard, §6.8.8

### Notes

Many users incorrectly attempt to use the construction

```
#ifdef __STDC__
```

instead of the correct form:

```
#if __STDC__
```

These constructions give different results because **__STDC__** is defined, but it is defined to a value of zero, in keeping with the fact that COHERENT C does not yet conform to the ANSI standard.

To help users avoid this error, COHERENT does not define **__STDC__** at all.

## ***__TIME__*** — Manifest Constant

Time source file is translated

**__TIME__** is a preprocessor constant that is defined by the C preprocessor. 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**.

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

### Example

For an example of how to use **__TIME__** in a program, see **__DATE__**.

### See Also

**__DATE__, __FILE__, __LINE__, __STDC__, manifest constant**
ANSI Standard §6.8.8

## ***_exit()*** — System Call (libc)

Terminate a program
**#include <unistd.h>**
**void _exit(*status*) int *status*;**

The system call **_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. For example, you may wish to call **_exit()** if 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. Unlike **exit()**, **_exit()** does not use STDIO. This will help you create programs that are extremely small when compiled.

### See Also

**close(), exit(), EXIT_FAILURE, EXIT_SUCCESS, libc, unistd.h, wait()**
POSIX Standard, §3.2.2

### Notes

If you do not explicitly set *status* to a value, the program returns whatever value happens to have been in the register EAX. You can set *status* to either **EXIT_SUCCESS** or **EXIT_FAILURE**.

## _getwd() — General Function (libc)

Get current working directory name
**char *_getwd(***pathname***)**
**char ***pathname***;**

The *current working directory* is the directory from which file name searches commence when a path name does not begin with '/'. **_getwd()** returns the name of the current working directory. It is useful for processes like spoolers and daemons, which must generate full path names for files.

If you do not have permission to search all levels of the directory hierarchy above the current directory, **_getwd()** cannot obtain the directory name for you.

### See Also

**chdir(), getcwd(), libc, pwd**

### Diagnostics

**_getwd()** returns NULL and writes an error message into *pathname* if an error occurs, e.g., if the current directory cannot be found or if any other error occurs.

### Notes

**_getwd()** is obsolete, and is included for reasons of compatibility. Programmers should use the function **getcwd()** instead.

**_getwd()** fails if the current directory name is longer than MAXPATH characters (128 characters as defined in header file **<path.h>**). The chunk of memory pointed to by *pathname* must be big enough to hold **MAXPATHLEN** characters plus a trailing NUL.

If **_getwd()** fails, the working directory cannot be restored to its initial value.

The name of this function has been change to **_getwd()** to avoid confusion with the Berkeley UNIX function **getwd()**, which has a different calling sequence.

## _tolower() — ctype Function (libc)

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

The function **_tolower()** converts the character *c* to lower case, and returns the converted character. Unlike the related function **tolower()**, **_tolower()** is not guaranteed to work correctly if handed anything other than an upper-case character, that is, a character for which **isupper()** returns true.

### See Also

**_toupper(), libc, tolower()**

### Notes

**_tolower()** is not part of the ANSI standard; COHERENT includes it only to support old code. You should use **tolower()** instead.

## _toupper() — ctype Function (libc)

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

The function **_toupper()** converts the character *c* to upper case and returns the converted character. Unlike the related function **toupper()**, **_toupper()** is not guaranteed to work correctly if it is passed something other than a lower-case character, that is, any character for which **islower()** returns true.

### See Also

**_tolower(), libc, toupper()**

### Notes

**_toupper()** is not part of the ANSI standard; COHERENT includes it only to support old code. You should use **toupper()** instead.