

# The C Preprocessor

---

For GCC version 16.0.0 (pre-release)

(GCC)

Richard M. Stallman, Zachary Weinberg

---

Copyright © 1987-2025 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled “GNU Free Documentation License”.

This manual contains no Invariant Sections. The Front-Cover Texts are (a) (see below), and the Back-Cover Texts are (b) (see below).

(a) The FSF’s Front-Cover Text is:

A GNU Manual

(b) The FSF’s Back-Cover Text is:

You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.

# Table of Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Overview</b>                              | <b>1</b>  |
| 1.1      | Character sets                               | 1         |
| 1.2      | Initial processing                           | 2         |
| 1.3      | Tokenization                                 | 4         |
| 1.4      | The preprocessing language                   | 6         |
| <b>2</b> | <b>Header Files</b>                          | <b>7</b>  |
| 2.1      | Include Syntax                               | 7         |
| 2.2      | Include Operation                            | 8         |
| 2.3      | Search Path                                  | 9         |
| 2.4      | Once-Only Headers                            | 9         |
| 2.5      | Alternatives to Wrapper <code>#ifndef</code> | 10        |
| 2.6      | Computed Includes                            | 10        |
| 2.7      | Wrapper Headers                              | 11        |
| 2.8      | System Headers                               | 12        |
| <b>3</b> | <b>Macros</b>                                | <b>13</b> |
| 3.1      | Object-like Macros                           | 13        |
| 3.2      | Function-like Macros                         | 14        |
| 3.3      | Macro Arguments                              | 15        |
| 3.4      | Stringizing                                  | 16        |
| 3.5      | Concatenation                                | 17        |
| 3.6      | Variadic Macros                              | 19        |
| 3.7      | Predefined Macros                            | 20        |
| 3.7.1    | Standard Predefined Macros                   | 20        |
| 3.7.2    | Common Predefined Macros                     | 22        |
| 3.7.3    | System-specific Predefined Macros            | 34        |
| 3.7.4    | C++ Named Operators                          | 34        |
| 3.8      | Undefining and Redefining Macros             | 35        |
| 3.9      | Directives Within Macro Arguments            | 36        |
| 3.10     | Macro Pitfalls                               | 36        |
| 3.10.1   | Misnesting                                   | 36        |
| 3.10.2   | Operator Precedence Problems                 | 36        |
| 3.10.3   | Swallowing the Semicolon                     | 37        |
| 3.10.4   | Duplication of Side Effects                  | 38        |
| 3.10.5   | Self-Referential Macros                      | 39        |
| 3.10.6   | Argument Prescan                             | 40        |
| 3.10.7   | Newlines in Arguments                        | 41        |
| <b>4</b> | <b>Conditionals</b>                          | <b>41</b> |
| 4.1      | Conditional Uses                             | 42        |
| 4.2      | Conditional Syntax                           | 42        |

|           |  |           |
|-----------|--|-----------|
| 4.2.1     | Ifdef.....                             | 42        |
| 4.2.2     | If.....                                | 43        |
| 4.2.3     | Defined.....                           | 44        |
| 4.2.4     | Else.....                              | 44        |
| 4.2.5     | Elif.....                              | 45        |
| 4.2.6     | __has_attribute.....                   | 45        |
| 4.2.7     | __has_cpp_attribute.....               | 46        |
| 4.2.8     | __has_c_attribute.....                 | 46        |
| 4.2.9     | __has_builtin.....                     | 46        |
| 4.2.10    | __has_feature.....                     | 46        |
| 4.2.11    | __has_extension.....                   | 47        |
| 4.2.12    | __has_include, __has_include_next..... | 47        |
| 4.2.13    | __has_embed.....                       | 48        |
| 4.3       | Deleted Code.....                      | 48        |
| <b>5</b>  | <b>Diagnostics .....</b>               | <b>48</b> |
| <b>6</b>  | <b>Line Control.....</b>               | <b>49</b> |
| <b>7</b>  | <b>Pragmas .....</b>                   | <b>50</b> |
| <b>8</b>  | <b>Binary Resource Inclusion.....</b>  | <b>51</b> |
| <b>9</b>  | <b>Other Directives .....</b>          | <b>52</b> |
| <b>10</b> | <b>Preprocessor Output .....</b>       | <b>53</b> |
| <b>11</b> | <b>Traditional Mode .....</b>          | <b>54</b> |
| 11.1      | Traditional lexical analysis .....     | 54        |
| 11.2      | Traditional macros .....               | 55        |
| 11.3      | Traditional miscellany .....           | 56        |
| 11.4      | Traditional warnings .....             | 56        |
| <b>12</b> | <b>Implementation Details .....</b>    | <b>57</b> |
| 12.1      | Implementation-defined behavior .....  | 57        |
| 12.2      | Implementation limits .....            | 58        |
| 12.3      | Obsolete Features .....                | 59        |
| 12.3.1    | Assertions .....                       | 59        |
| <b>13</b> | <b>Invocation .....</b>                | <b>60</b> |
| <b>14</b> | <b>Environment Variables .....</b>     | <b>70</b> |

|  |           |
|--|-----------|
| <b>GNU Free Documentation License .....</b>                | <b>72</b> |
| ADDENDUM: How to use this License for your documents ..... | 79        |
| <b>Index of Directives .....</b>                           | <b>80</b> |
| <b>Option Index .....</b>                                  | <b>80</b> |
| <b>Concept Index .....</b>                                 | <b>82</b> |

# 1 Overview

The C preprocessor, often known as *cpp*, is a *macro processor* that is used automatically by the C compiler to transform your program before compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor is intended to be used only with C, C++, and Objective-C source code. In the past, it has been abused as a general text processor. It will choke on input which does not obey C's lexical rules. For example, apostrophes will be interpreted as the beginning of character constants, and cause errors. Also, you cannot rely on it preserving characteristics of the input which are not significant to C-family languages. If a Makefile is preprocessed, all the hard tabs will be removed, and the Makefile will not work.

Having said that, you can often get away with using *cpp* on things which are not C. Other Algol-ish programming languages are often safe (Ada, etc.) So is assembly, with caution. `-traditional-cpp` mode preserves more white space, and is otherwise more permissive. Many of the problems can be avoided by writing C or C++ style comments instead of native language comments, and keeping macros simple.

Wherever possible, you should use a preprocessor geared to the language you are writing in. Modern versions of the GNU assembler have macro facilities. Most high level programming languages have their own conditional compilation and inclusion mechanism. If all else fails, try a true general text processor, such as GNU M4.

C preprocessors vary in some details. This manual discusses the GNU C preprocessor, which provides a small superset of the features of ISO Standard C. In its default mode, the GNU C preprocessor does not do a few things required by the standard. These are features which are rarely, if ever, used, and may cause surprising changes to the meaning of a program which does not expect them. To get strict ISO Standard C, you should use the `-std=c90`, `-std=c99`, `-std=c11` or `-std=c17` options, depending on which version of the standard you want. To get all the mandatory diagnostics, you must also use `-pedantic`. See Chapter 13 [Invocation], page 60.

This manual describes the behavior of the ISO preprocessor. To minimize gratuitous differences, where the ISO preprocessor's behavior does not conflict with traditional semantics, the traditional preprocessor should behave the same way. The various differences that do exist are detailed in the section Chapter 11 [Traditional Mode], page 54.

For clarity, unless noted otherwise, references to 'CPP' in this manual refer to GNU CPP.

## 1.1 Character sets

Source code character set processing in C and related languages is rather complicated. The C standard discusses two character sets, but there are really at least four.

The files input to CPP might be in any character set at all. CPP's very first action, before it even looks for line boundaries, is to convert the file into the character set it uses for internal processing. That set is what the C standard calls the *source* character set. It must be isomorphic with ISO 10646, also known as Unicode. CPP uses the UTF-8 encoding of Unicode.

The character sets of the input files are specified using the `-finput-charset=` option.

All preprocessing work (the subject of the rest of this manual) is carried out in the source character set. If you request textual output from the preprocessor with the `-E` option, it will be in UTF-8.

After preprocessing is complete, string and character constants are converted again, into the *execution* character set. This character set is under control of the user; the default is UTF-8, matching the source character set. Wide string and character constants have their own character set, which is not called out specifically in the standard. Again, it is under control of the user. The default is UTF-16 or UTF-32, whichever fits in the target’s `wchar_t` type, in the target machine’s byte order.<sup>1</sup> Octal and hexadecimal escape sequences do not undergo conversion; `‘\x12’` has the value 0x12 regardless of the currently selected execution character set. All other escapes are replaced by the character in the source character set that they represent, then converted to the execution character set, just like unescaped characters.

In identifiers, characters outside the ASCII range can be specified with the `‘\u’` and `‘\U’` escapes or used directly in the input encoding. If strict ISO C90 conformance is specified with an option such as `-std=c90`, or `-fno-extended-identifiers` is used, then those constructs are not permitted in identifiers.

## 1.2 Initial processing

The preprocessor performs a series of textual transformations on its input. These happen before all other processing. Conceptually, they happen in a rigid order, and the entire file is run through each transformation before the next one begins. CPP actually does them all at once, for performance reasons. These transformations correspond roughly to the first three “phases of translation” described in the C standard.

1. The input file is read into memory and broken into lines.

Different systems use different conventions to indicate the end of a line. GCC accepts the ASCII control sequences *LF*, *CR LF* and *CR* as end-of-line markers. These are the canonical sequences used by Unix, DOS and VMS, and the classic Mac OS (before OSX) respectively. You may therefore safely copy source code written on any of those systems to a different one and use it without conversion. (GCC may lose track of the current line number if a file doesn’t consistently use one convention, as sometimes happens when it is edited on computers with different conventions that share a network file system.)

If the last line of any input file lacks an end-of-line marker, the end of the file is considered to implicitly supply one. The C standard says that this condition provokes undefined behavior, so GCC will emit a warning message.

2. If trigraphs are enabled, they are replaced by their corresponding single characters. By default GCC ignores trigraphs, but if you request a strictly conforming mode with the `-std` option, or you specify the `-trigraphs` option, then it converts them.

These are nine three-character sequences, all starting with `‘??’`, that are defined by ISO C to stand for single characters. They permit obsolete systems that lack some of

---

<sup>1</sup> UTF-16 does not meet the requirements of the C standard for a wide character set, but the choice of 16-bit `wchar_t` is enshrined in some system ABIs so we cannot fix this.



Line comments are not in the 1989 edition of the C standard, but they are recognized by GCC as an extension. In C++ and in the 1999 edition of the C standard, they are an official part of the language.

Since these transformations happen before all other processing, you can split a line mechanically with backslash-newline anywhere. You can comment out the end of a line. You can continue a line comment onto the next line with backslash-newline. You can even split `/*`, `*/`, and `/**` onto multiple lines with backslash-newline. For example:

```
/\
*
*/ # /*
*/ defi\
ne F0\
0 10\
20
```

is equivalent to `#define F00 1020`. All these tricks are extremely confusing and should not be used in code intended to be readable.

There is no way to prevent a backslash at the end of a line from being interpreted as a backslash-newline. This cannot affect any correct program, however.

### 1.3 Tokenization

After the textual transformations are finished, the input file is converted into a sequence of *preprocessing tokens*. These mostly correspond to the syntactic tokens used by the C compiler, but there are a few differences. White space separates tokens; it is not itself a token of any kind. Tokens do not have to be separated by white space, but it is often necessary to avoid ambiguities.

When faced with a sequence of characters that has more than one possible tokenization, the preprocessor is greedy. It always makes each token, starting from the left, as big as possible before moving on to the next token. For instance, `a+++++b` is interpreted as `a ++ ++ + b`, not as `a ++ + ++ b`, even though the latter tokenization could be part of a valid C program and the former could not.

Once the input file is broken into tokens, the token boundaries never change, except when the `##` preprocessing operator is used to paste tokens together. See Section 3.5 [Concatenation], page 17. For example,

```
#define foo() bar
foo()baz
    ↪ bar baz
not
    ↪ barbaz
```

The compiler does not re-tokenize the preprocessor's output. Each preprocessing token becomes one compiler token.

Preprocessing tokens fall into five broad classes: identifiers, preprocessing numbers, string literals, punctuators, and other. An *identifier* is the same as an identifier in C: any sequence of letters, digits, or underscores, which begins with a letter or underscore. Keywords of C have no significance to the preprocessor; they are ordinary identifiers. You can define a macro whose name is a keyword, for instance. The only identifier which can be considered a preprocessing keyword is `defined`. See Section 4.2.3 [Defined], page 44.





*directive name*. It specifies the operation to perform. Directives are commonly referred to as `#name` where *name* is the directive name. For example, `#define` is the directive that defines a macro.

The `#` which begins a directive cannot come from a macro expansion. Also, the directive name is not macro expanded. Thus, if `foo` is defined as a macro expanding to `define`, that does not make `#foo` a valid preprocessing directive.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives.

Some directives require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, `#define` must be followed by a macro name and the intended expansion of the macro.

A preprocessing directive cannot cover more than one line. The line may, however, be continued with backslash-newline, or by a block comment which extends past the end of the line. In either case, when the directive is processed, the continuations have already been merged with the first line to make one long line.

## 2 Header Files

A header file is a file containing C declarations and macro definitions (see Chapter 3 [Macros], page 13) to be shared between several source files. You request the use of a header file in your program by *including* it, with the C preprocessing directive `#include`.

Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

In C, the usual convention is to give header files names that end with `.h`. It is most portable to use only letters, digits, dashes, and underscores in header file names, and at most one dot.

### 2.1 Include Syntax

Both user and system header files are included using the preprocessing directive `#include`. It has two variants:

**#include <file>**

This variant is used for system header files. It searches for a file named *file* in a standard list of system directories. You can prepend directories to this list with the `-I` option (see Chapter 13 [Invocation], page 60).

**#include "file"**

This variant is used for header files of your own program. It searches for a file named *file* first in the directory containing the current file, then in the quote directories and then the same directories used for `<file>`. You can prepend directories to the list of quote directories with the `-iquote` option.

The argument of `#include`, whether delimited with quote marks or angle brackets, behaves like a string constant in that comments are not recognized, and macro names are not expanded. Thus, `#include <x/*y>` specifies inclusion of a system header file named `x/*y`.

However, if backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"` specifies a filename containing three backslashes. (Some systems interpret `'\'` as a pathname separator. All of these also interpret `'/'` the same way. It is most portable to use only `'/'`.)

It is an error if there is anything (other than comments) on the line after the file name.

## 2.2 Include Operation

The `#include` directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the `#include` directive. For example, if you have a header file `header.h` as follows,

```
char *test (void);
```

and a main program called `program.c` that uses the header file, like this,

```
int x;
#include "header.h"

int
main (void)
{
    puts (test ());
}
```

the compiler will see the same token stream as it would if `program.c` read

```
int x;
char *test (void);

int
main (void)
{
    puts (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include



```

/* File foo. */
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

    the entire file

#endif /* !FILE_FOO_SEEN */

```

This construct is commonly known as a *wrapper* `#ifndef`. When the header is included again, the conditional will be false, because `FILE_FOO_SEEN` is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

CPP optimizes even further. It remembers when a header file has a wrapper `'#ifndef'`. If a subsequent `'#include'` specifies that header, and the macro in the `'#ifndef'` is still defined, it does not bother to rescan the file at all.

You can put comments outside the wrapper. They will not interfere with this optimization.

The macro `FILE_FOO_SEEN` is called the *controlling macro* or *guard macro*. In a user header file, the macro name should not begin with `'_'`. In a system header file, it should begin with `'__'` to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

## 2.5 Alternatives to Wrapper `#ifndef`

CPP supports two more ways of indicating that a header file should be read only once. Neither one is as portable as a wrapper `'#ifndef'` and we recommend you do not use them in new programs, with the caveat that `'#import'` is standard practice in Objective-C.

CPP supports a variant of `'#include'` called `'#import'` which includes a file, but does so at most once. If you use `'#import'` instead of `'#include'`, then you don't need the conditionals inside the header file to prevent multiple inclusion of the contents. `'#import'` is standard in Objective-C, but is considered a deprecated extension in C and C++.

`'#import'` is not a well designed feature. It requires the users of a header file to know that it should only be included once. It is much better for the header file's implementor to write the file so that users don't need to know this. Using a wrapper `'#ifndef'` accomplishes this goal.

In the present implementation, a single use of `'#import'` will prevent the file from ever being read again, by either `'#import'` or `'#include'`. You should not rely on this; do not use both `'#import'` and `'#include'` to refer to the same header file.

Another way to prevent a header file from being included more than once is with the `'#pragma once'` directive (see Chapter 7 [Pragmas], page 50). `'#pragma once'` does not have the problems that `'#import'` does, but it is not recognized by all preprocessors, so you cannot rely on it in a portable program.

## 2.6 Computed Includes

Sometimes it is necessary to select one of several different header files to be included into your program. They might specify configuration parameters to be used on different sorts of operating systems, for instance. You could do this with a series of conditionals,

```

#if SYSTEM_1

```





- There is also a directive, `#pragma GCC system_header`, which tells GCC to consider the rest of the current include file a system header, no matter where it was found. Code that comes before the `#pragma` in the file is not affected. `#pragma GCC system_header` has no effect in the primary source file.

On some targets, such as RS/6000 AIX, GCC implicitly surrounds all system headers with an `'extern "C"'` block when compiling as C++.

## 3 Macros

A *macro* is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. *Object-like* macros resemble data objects when used, *function-like* macros resemble function calls.

You may define any valid identifier as a macro, even if it is a C keyword. The preprocessor does not know anything about keywords. This can be useful if you wish to hide a keyword such as `const` from an older compiler that does not understand it. However, the preprocessor operator `defined` (see Section 4.2.3 [Defined], page 44) can never be defined as a macro, and C++'s named operators (see Section 3.7.4 [C++ Named Operators], page 34) cannot be macros when you are compiling C++.

### 3.1 Object-like Macros

An *object-like macro* is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it. They are most commonly used to give symbolic names to numeric constants.

You create macros with the `'#define'` directive. `'#define'` is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's *body*, *expansion* or *replacement list*. For example,

```
#define BUFFER_SIZE 1024
```

defines a macro named `BUFFER_SIZE` as an abbreviation for the token `1024`. If somewhere after this `'#define'` directive there comes a C statement of the form

```
foo = (char *) malloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro `BUFFER_SIZE`. The C compiler will see the same tokens as it would if you had written

```
foo = (char *) malloc (1024);
```

By convention, macro names are written in uppercase. Programs are easier to read when it is possible to tell at a glance which names are macros.

The macro's body ends at the end of the `'#define'` line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

```
#define NUMBERS 1, \
                2, \
                3
int x[] = { NUMBERS };
↪ int x[] = { 1, 2, 3 };
```







Here is an example of a macro definition that uses stringizing:

```
#define WARN_IF(EXP) \
do { if (EXP) \
    fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
WARN_IF (x == 0);
    ↳ do { if (x == 0)
        fprintf (stderr, "Warning: " "x == 0" "\n"); } while (0);
```

The argument for `EXP` is substituted once, as-is, into the `if` statement, and once, stringized, into the argument to `fprintf`. If `x` were a macro, it would be expanded in the `if` statement, but not in the string.

The `do` and `while (0)` are a kludge to make it possible to write `WARN_IF (arg);`, which the resemblance of `WARN_IF` to a function would make C programmers want to do; see Section 3.10.3 [Swallowing the Semicolon], page 37.

Stringizing in C involves more than putting double-quote characters around the fragment. The preprocessor backslash-escapes the quotes surrounding embedded string constants, and all backslashes within string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringizing `p = "foo\n"`; results in `"p = \"foo\\n\""`. However, backslashes that are not inside string or character constants are not duplicated: `'\n'` by itself stringizes to `"\n"`.

All leading and trailing whitespace in text being stringized is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringized result. Comments are replaced by whitespace long before stringizing happens, so they never appear in stringized text.

There is no way to convert a macro argument into a character constant.

If you want to stringize the result of expansion of a macro argument, you have to use two levels of macros.

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
str (foo)
    ↳ "foo"
xstr (foo)
    ↳ xstr (4)
    ↳ str (4)
    ↳ "4"
```

`s` is stringized when it is used in `str`, so it is not macro-expanded first. But `s` is an ordinary argument to `xstr`, so it is completely macro-expanded before `xstr` itself is expanded (see Section 3.10.6 [Argument Prescan], page 40). Therefore, by the time `str` gets to its argument, it has already been macro-expanded.

## 3.5 Concatenation

It is often useful to merge two tokens into one while expanding macros. This is called *token pasting* or *token concatenation*. The `##` preprocessing operator performs token pasting. When a macro is expanded, the two tokens on either side of each `##` operator are combined into a single token, which then replaces the `##` and the two original tokens in the macro expansion. Usually both will be identifiers, or one will be an identifier and the other a

preprocessing number. When pasted, they make a longer identifier. This isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as 1.5 and e3) into a number. Also, multi-character operators such as += can be formed by token pasting.

However, two tokens that don't together form a valid token cannot be pasted together. For example, you cannot concatenate `x` with `+` in either order. If you try, the preprocessor issues a warning and emits the two tokens. Whether it puts white space between the tokens is undefined. It is common to find unnecessary uses of `##` in complex macros. If you get this warning, it is likely that you can simply remove the `##`.

Both the tokens combined by `##` could come from the macro body, but you could just as well write them as one token in the first place. Token pasting is most useful when one or both of the tokens comes from a macro argument. If either of the tokens next to an `##` is a parameter name, it is replaced by its actual argument before `##` executes. As with stringizing, the actual argument is not macro-expanded first. If the argument is empty, that `##` has no effect.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating `/'` and `*`. You can put as much whitespace between `##` and its operands as you like, including comments, and you can put comments in arguments that will be concatenated. However, it is an error if `##` appears at either end of a macro body.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
    char *name;
    void (*function) (void);
};

struct command commands[] =
{
    { "quit", quit_command },
    { "help", help_command },
    ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringizing, and the function name by concatenating the argument with `_command`. Here is how it is done:

```
#define COMMAND(NAME) { #NAME, NAME ## _command }

struct command commands[] =
{
    COMMAND (quit),
    COMMAND (help),
    ...
};
```













**\_\_REGISTER\_PREFIX\_\_**

This macro expands to a single token (not a string constant) which is the prefix applied to CPU register names in assembly language for this target. You can use it to write assembly that is usable in multiple environments. For example, in the **m68k-aout** environment it expands to nothing, but in the **m68k-coff** environment it expands to a single **'%'**.

**\_\_USER\_LABEL\_PREFIX\_\_**

This macro expands to a single token which is the prefix applied to user labels (symbols visible to C code) in assembly. For example, in the **m68k-aout** environment it expands to an **'\_'**, but in the **m68k-coff** environment it expands to nothing.

This macro will have the correct definition even if **-f(no-)underscores** is in use, but it will not be correct if target-specific options that adjust this prefix are used (e.g. the OSF/rose **-mno-underscores** option).

```

__SIZE_TYPE__
__PTRDIFF_TYPE__
__WCHAR_TYPE__
__WINT_TYPE__
__INTMAX_TYPE__
__UINTMAX_TYPE__
__SIG_ATOMIC_TYPE__
__INT8_TYPE__
__INT16_TYPE__
__INT32_TYPE__
__INT64_TYPE__
__UINT8_TYPE__
__UINT16_TYPE__
__UINT32_TYPE__
__UINT64_TYPE__
__INT_LEAST8_TYPE__
__INT_LEAST16_TYPE__
__INT_LEAST32_TYPE__
__INT_LEAST64_TYPE__
__UINT_LEAST8_TYPE__
__UINT_LEAST16_TYPE__
__UINT_LEAST32_TYPE__
__UINT_LEAST64_TYPE__
__INT_FAST8_TYPE__
__INT_FAST16_TYPE__
__INT_FAST32_TYPE__
__INT_FAST64_TYPE__
__UINT_FAST8_TYPE__
__UINT_FAST16_TYPE__
__UINT_FAST32_TYPE__
__UINT_FAST64_TYPE__
__INTPTR_TYPE__
__UINTPTR_TYPE__

```

These macros are defined to the correct underlying types for the `size_t`, `ptrdiff_t`, `wchar_t`, `wint_t`, `intmax_t`, `uintmax_t`, `sig_atomic_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int_least8_t`, `int_least16_t`, `int_least32_t`, `int_least64_t`, `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, `uint_least64_t`, `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `int_fast64_t`, `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, `uint_fast64_t`, `intptr_t`, and `uintptr_t` typedefs, respectively. They exist to make the standard header files `stddef.h`, `stdint.h`, and `wchar.h` work correctly. You should not use these macros directly; instead, include the appropriate headers and use the typedefs. Some of these macros may not be defined on particular systems if GCC does not provide a `stdint.h` header on those systems.

**\_\_CHAR\_BIT\_\_**

Defined to the number of bits used in the representation of the `char` data type. It exists to make the standard header given numerical limits work correctly. You should not use this macro directly; instead, include the appropriate headers.

```

__SCHAR_MAX__
__WCHAR_MAX__
__SHRT_MAX__
__INT_MAX__
__LONG_MAX__
__LONG_LONG_MAX__
__WINT_MAX__
__SIZE_MAX__
__PTRDIFF_MAX__
__INTMAX_MAX__
__UINTMAX_MAX__
__SIG_ATOMIC_MAX__
__INT8_MAX__
__INT16_MAX__
__INT32_MAX__
__INT64_MAX__
__UINT8_MAX__
__UINT16_MAX__
__UINT32_MAX__
__UINT64_MAX__
__INT_LEAST8_MAX__
__INT_LEAST16_MAX__
__INT_LEAST32_MAX__
__INT_LEAST64_MAX__
__UINT_LEAST8_MAX__
__UINT_LEAST16_MAX__
__UINT_LEAST32_MAX__
__UINT_LEAST64_MAX__
__INT_FAST8_MAX__
__INT_FAST16_MAX__
__INT_FAST32_MAX__
__INT_FAST64_MAX__
__UINT_FAST8_MAX__
__UINT_FAST16_MAX__
__UINT_FAST32_MAX__
__UINT_FAST64_MAX__
__INTPTR_MAX__
__UINTPTR_MAX__
__WCHAR_MIN__
__WINT_MIN__
__SIG_ATOMIC_MIN__

```

Defined to the maximum value of the signed char, wchar\_t, signed short, signed int, signed long, signed long long, wint\_t, size\_t, ptrdiff\_t, intmax\_t, uintmax\_t, sig\_atomic\_t, int8\_t, int16\_t, int32\_t, int64\_t, uint8\_t, uint16\_t, uint32\_t, uint64\_t, int\_least8\_t, int\_least16\_t, int\_least32\_t, int\_least64\_t, uint\_least8\_t, uint\_least16\_t, uint\_least32\_t, uint\_least64\_t, int\_fast8\_t, int\_fast16\_t, int\_

`fast32_t`, `int_fast64_t`, `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, `uint_fast64_t`, `intptr_t`, and `uintptr_t` types and to the minimum value of the `wchar_t`, `wint_t`, and `sig_atomic_t` types respectively. They exist to make the standard header given numerical limits work correctly. You should not use these macros directly; instead, include the appropriate headers. Some of these macros may not be defined on particular systems if GCC does not provide a `stdint.h` header on those systems.

```
__INT8_C
__INT16_C
__INT32_C
__INT64_C
__UINT8_C
__UINT16_C
__UINT32_C
__UINT64_C
__INTMAX_C
__UINTMAX_C
```

Defined to implementations of the standard `stdint.h` macros with the same names without the leading `__`. They exist to make the implementation of that header work correctly. You should not use these macros directly; instead, include the appropriate headers. Some of these macros may not be defined on particular systems if GCC does not provide a `stdint.h` header on those systems.

```
__SCHAR_WIDTH__
__SHRT_WIDTH__
__INT_WIDTH__
__LONG_WIDTH__
__LONG_LONG_WIDTH__
__PTRDIFF_WIDTH__
__SIG_ATOMIC_WIDTH__
__SIZE_WIDTH__
__WCHAR_WIDTH__
__WINT_WIDTH__
__INT_LEAST8_WIDTH__
__INT_LEAST16_WIDTH__
__INT_LEAST32_WIDTH__
__INT_LEAST64_WIDTH__
__INT_FAST8_WIDTH__
__INT_FAST16_WIDTH__
__INT_FAST32_WIDTH__
__INT_FAST64_WIDTH__
__INTPTR_WIDTH__
__INTMAX_WIDTH__
```

Defined to the bit widths of the corresponding types. They exist to make the implementations of `limits.h` and `stdint.h` behave correctly. You should not use these macros directly; instead, include the appropriate headers. Some of

these macros may not be defined on particular systems if GCC does not provide a `stdint.h` header on those systems.

```
__SIZEOF_INT__
__SIZEOF_LONG__
__SIZEOF_LONG_LONG__
__SIZEOF_SHORT__
__SIZEOF_POINTER__
__SIZEOF_FLOAT__
__SIZEOF_DOUBLE__
__SIZEOF_LONG_DOUBLE__
__SIZEOF_SIZE_T__
__SIZEOF_WCHAR_T__
__SIZEOF_WINT_T__
__SIZEOF_PTRDIFF_T__
```

Defined to the number of bytes of the C standard data types: `int`, `long`, `long long`, `short`, `void *`, `float`, `double`, `long double`, `size_t`, `wchar_t`, `wint_t` and `ptrdiff_t`.

```
__BYTE_ORDER__
__ORDER_LITTLE_ENDIAN__
__ORDER_BIG_ENDIAN__
__ORDER_PDP_ENDIAN__
```

`__BYTE_ORDER__` is defined to one of the values `__ORDER_LITTLE_ENDIAN__`, `__ORDER_BIG_ENDIAN__`, or `__ORDER_PDP_ENDIAN__` to reflect the layout of multi-byte and multi-word quantities in memory. If `__BYTE_ORDER__` is equal to `__ORDER_LITTLE_ENDIAN__` or `__ORDER_BIG_ENDIAN__`, then multi-byte and multi-word quantities are laid out identically: the byte (word) at the lowest address is the least significant or most significant byte (word) of the quantity, respectively. If `__BYTE_ORDER__` is equal to `__ORDER_PDP_ENDIAN__`, then bytes in 16-bit words are laid out in a little-endian fashion, whereas the 16-bit subwords of a 32-bit quantity are laid out in big-endian fashion.

You should use these macros for testing like this:

```
/* Test for a little-endian machine */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
```

```
__FLOAT_WORD_ORDER__
```

`__FLOAT_WORD_ORDER__` is defined to one of the values `__ORDER_LITTLE_ENDIAN__` or `__ORDER_BIG_ENDIAN__` to reflect the layout of the words of multi-word floating-point quantities.

```
__DEPRECATED
```

This macro is defined, with value 1, when compiling a C++ source file with warnings about deprecated constructs enabled. These warnings are enabled by default, but can be disabled with `-Wno-deprecated`.

```
__EXCEPTIONS
```

This macro is defined, with value 1, when compiling a C++ source file with exceptions enabled. If `-fno-exceptions` is used when compiling the file, then this macro is not defined.





**\_\_GCC\_IEC\_559**

This macro is defined to indicate the intended level of support for IEEE 754 (IEC 60559) floating-point arithmetic. It expands to a nonnegative integer value. If 0, it indicates that the combination of the compiler configuration and the command-line options is not intended to support IEEE 754 arithmetic for `float` and `double` as defined in C99 and C11 Annex F (for example, that the standard rounding modes and exceptions are not supported, or that optimizations are enabled that conflict with IEEE 754 semantics). If 1, it indicates that IEEE 754 arithmetic is intended to be supported; this does not mean that all relevant language features are supported by GCC. If 2 or more, it additionally indicates support for IEEE 754-2008 (in particular, that the binary encodings for quiet and signaling NaNs are as specified in IEEE 754-2008).

This macro does not indicate the default state of command-line options that control optimizations that C99 and C11 permit to be controlled by standard pragmas, where those standards do not require a particular default state. It does not indicate whether optimizations respect signaling NaN semantics (the macro for that is `__SUPPORT_SNAN__`). It does not indicate support for decimal floating point or the IEEE 754 binary16 and binary128 types.

**\_\_GCC\_IEC\_559\_COMPLEX**

This macro is defined to indicate the intended level of support for IEEE 754 (IEC 60559) floating-point arithmetic for complex numbers, as defined in C99 and C11 Annex G. It expands to a nonnegative integer value. If 0, it indicates that the combination of the compiler configuration and the command-line options is not intended to support Annex G requirements (for example, because `-fcx-limited-range` was used). If 1 or more, it indicates that it is intended to support those requirements; this does not mean that all relevant language features are supported by GCC.

**\_\_NO\_MATH\_ERRNO\_\_**

This macro is defined if `-fno-math-errno` is used, or enabled by another option such as `-ffast-math` or by default.

**\_\_RECIPROCAL\_MATH\_\_**

This macro is defined if `-freciprocal-math` is used, or enabled by another option such as `-ffast-math` or by default.

**\_\_NO\_SIGNED\_ZEROS\_\_**

This macro is defined if `-fno-signed-zeros` is used, or enabled by another option such as `-ffast-math` or by default.

**\_\_NO\_TRAPPING\_MATH\_\_**

This macro is defined if `-fno-trapping-math` is used.

**\_\_ASSOCIATIVE\_MATH\_\_**

This macro is defined if `-fassociative-math` is used, or enabled by another option such as `-ffast-math` or by default.

**\_\_ROUNDING\_MATH\_\_**

This macro is defined if `-frounding-math` is used.















`lose` requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
or
#define bar(x) lose((x))
```

The extra pair of parentheses prevents the comma in `foo`'s definition from being interpreted as an argument separator.

### 3.10.7 Newlines in Arguments

The invocation of a function-like macro can extend over many logical lines. However, in the present implementation, the entire expansion comes out on one line. Thus line numbers emitted by the compiler or debugger refer to the line the invocation started on, which might be different to the line containing the argument causing the problem.

Here is an example illustrating this:

```
#define ignore_second_arg(a,b,c) a; c

ignore_second_arg (foo (),
                  ignored (),
                  syntax error);
```

The syntax error triggered by the tokens `syntax error` results in an error message citing line three—the line of `ignore_second_arg`—even though the problematic code comes from line five.

We consider this a bug, and intend to fix it in the near future.

## 4 Conditionals

A *conditional* is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special `defined` operator.

A conditional in the C preprocessor resembles in some ways an `if` statement in C, but it is important to understand the difference between them. The condition in an `if` statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessing conditional directive is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

However, the distinction is becoming less clear. Modern compilers often do test `if` statements when a program is compiled, if their conditions are known not to vary at run time, and eliminate code which can never be executed. If you can count on your compiler to do this, you may find that your program is more readable if you use `if` statements with constant conditions (perhaps determined by macros). Of course, you can only use this to exclude code, not type definitions or other preprocessing directives, and you can only do it if the code remains syntactically valid when it is not to be used.









expression as shown below would only be valid with a compiler that supports the operator but not with others that don't.

```
#if defined __has_attribute && __has_attribute (nonnull)    /* not portable */
...
#endif
```

### 4.2.7 `__has_cpp_attribute`

The special operator `__has_cpp_attribute (operand)` may be used in `'#if'` and `'#elif'` expressions in C++ code to test whether the attribute referenced by its *operand* is recognized by GCC. `__has_cpp_attribute (operand)` is equivalent to `__has_attribute (operand)` except that when *operand* designates a supported standard attribute it evaluates to an integer constant of the form YYYYMM indicating the year and month when the attribute was first introduced into the C++ standard. For additional information including the dates of the introduction of current standard attributes, see SD-6: SG10 Feature Test Recommendations (<https://isocpp.org/std/standing-documents/sd-6-sg10-f>)

### 4.2.8 `__has_c_attribute`

The special operator `__has_c_attribute (operand)` may be used in `'#if'` and `'#elif'` expressions in C code to test whether the attribute referenced by its *operand* is recognized by GCC in attributes using the `'[[]]'` syntax. GNU attributes must be specified with the scope `'gnu'` or `'__gnu__'` with `__has_c_attribute`. When *operand* designates a supported standard attribute it evaluates to an integer constant of the form YYYYMM indicating the year and month when the attribute was first introduced into the C standard, or when the syntax of operands to the attribute was extended in the C standard.

### 4.2.9 `__has_builtin`

The special operator `__has_builtin (operand)` may be used in constant integer contexts and in preprocessor `'#if'` and `'#elif'` expressions to test whether the symbol named by its *operand* is recognized as a built-in function by GCC in the current language and conformance mode. It evaluates to a constant integer with a nonzero value if the argument refers to such a function, and to zero otherwise. The operator may also be used in preprocessor `'#if'` and `'#elif'` expressions. The `__has_builtin` operator by itself, without any *operand* or parentheses, acts as a predefined macro so that support for it can be tested in portable code. Thus, the recommended use of the operator is as follows:

```
#if defined __has_builtin
# if __has_builtin (__builtin_object_size)
#   define builtin_object_size(ptr) __builtin_object_size (ptr, 2)
# endif
#endif
#ifndef builtin_object_size
# define builtin_object_size(ptr) ((size_t)-1)
#endif
```

### 4.2.10 `__has_feature`

The special operator `__has_feature (operand)` may be used in constant integer contexts and in preprocessor `'#if'` and `'#elif'` expressions to test whether the identifier given in *operand* is recognized as a feature supported by GCC given the current options and, in the

















Generally speaking, in traditional mode an opening quote need not have a matching closing quote. In particular, a macro may be defined with replacement text that contains an unmatched quote. Of course, if you attempt to compile preprocessed output containing an unmatched quote you will get a syntax error.

However, all preprocessing directives other than `#define` require matching quotes. For example:

```
#define m This macro's fine and has an unmatched quote
/* This is not a comment. */
/* This is a comment. The following #include directive
   is ill-formed. */
#include <stdio.h>
```

Just as for the ISO preprocessor, what would be a closing quote can be escaped with a backslash to prevent the quoted text from closing.

## 11.2 Traditional macros

The major difference between traditional and ISO macros is that the former expand to text rather than to a token sequence. CPP removes all leading and trailing horizontal whitespace from a macro's replacement text before storing it, but preserves the form of internal whitespace.

One consequence is that it is legitimate for the replacement text to contain an unmatched quote (see Section 11.1 [Traditional lexical analysis], page 54). An unclosed string or character constant continues into the text following the macro call. Similarly, the text at the end of a macro's expansion can run together with the text after the macro invocation to produce a single token.

Normally comments are removed from the replacement text after the macro is expanded, but if the `-CC` option is passed on the command-line comments are preserved. (In fact, the current implementation removes comments even before saving the macro replacement text, but it careful to do it in such a way that the observed effect is identical even in the function-like macro case.)

The ISO stringizing operator `#` and token paste operator `##` have no special meaning. As explained later, an effect similar to these operators can be obtained in a different way. Macro names that are embedded in quotes, either from the main file or after macro replacement, do not expand.

CPP replaces an unquoted object-like macro name with its replacement text, and then rescans it for further macros to replace. Unlike standard macro expansion, traditional macro expansion has no provision to prevent recursion. If an object-like macro appears unquoted in its replacement text, it will be replaced again during the rescan pass, and so on *ad infinitum*. GCC detects when it is expanding recursive macros, emits an error message, and continues after the offending macro invocation.

```
#define PLUS +
#define INC(x) PLUS+x
INC(foo);
    ↦ ++foo;
```

Function-like macros are similar in form but quite different in behavior to their ISO counterparts. Their arguments are contained within parentheses, are comma-separated, and can cross physical lines. Commas within nested parentheses are not treated as argument



Presently `-Wtraditional` warns about:

- Macro parameters that appear within string literals in the macro body. In traditional C macro replacement takes place within string literals, but does not in ISO C.
- In traditional C, some preprocessor directives did not exist. Traditional preprocessors would only consider a line to be a directive if the `#` appeared in column 1 on the line. Therefore `-Wtraditional` warns about directives that traditional C understands but would ignore because the `#` does not appear as the first character on the line. It also suggests you hide directives like `#pragma` not understood by traditional C by indenting them. Some traditional implementations would not recognize `#elif`, so it suggests avoiding it altogether.
- A function-like macro that appears without an argument list. In some traditional preprocessors this was an error. In ISO C it merely means that the macro is not expanded.
- The unary plus operator. This did not exist in traditional C.
- The `'U'` and `'LL'` integer constant suffixes, which were not available in traditional C. (Traditional C does support the `'L'` suffix for simple long integer constants.) You are not warned about uses of these suffixes in macros defined in system headers. For instance, `UINT_MAX` may well be defined as `4294967295U`, but you will not be warned if you use `UINT_MAX`.

You can usually avoid the warning, and the related warning about constants which are so large that they are unsigned, by writing the integer constant in question in hexadecimal, with no U suffix. Take care, though, because this gives the wrong result in exotic cases.

## 12 Implementation Details

Here we document details of how the preprocessor's implementation affects its user-visible behavior. You should try to avoid undue reliance on behavior described here, as it is possible that it will change subtly in future implementations.

Also documented here are obsolete features still supported by CPP.

### 12.1 Implementation-defined behavior

This is how CPP behaves in all the cases which the C standard describes as *implementation-defined*. This term means that the implementation is free to do what it likes, but must document its choice and stick to it.

- The mapping of physical source file multi-byte characters to the execution character set.

The input character set can be specified using the `-finput-charset` option, while the execution character set may be controlled using the `-fexec-charset` and `-fwide-exec-charset` options.

- Identifier characters.

The C and C++ standards allow identifiers to be composed of `'_'` and the alphanumeric characters. C++ also allows universal character names. C99 and later C standards































under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING



- be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
  - C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
  - D. Preserve all the copyright notices of the Document.
  - E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
  - F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
  - G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
  - H. Include an unaltered copy of this License.
  - I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
  - J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
  - K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
  - L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
  - M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
  - N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
  - O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <https://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

## 11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.







