Frexx C Preprocessor

```
                    The  Frexx  C  Preprocessor
                    ========================
```

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

   The C preprocessor provides four separate facilities that you can use as you
see fit:

   * Inclusion of header files. These are files of declarations that
     can be substituted into your program.

   * Macro expansion. You can define "macros", which are abbreviations
     for arbitrary fragments of C code, and then the C preprocessor will
     replace the macros with their definitions throughout the program.

   * Conditional compilation. Using special preprocessing directives,
     you can include or exclude parts of the program according to
     various conditions.

   * Line control. If you use a program to combine or rearrange source
     files into an intermediate file which is then compiled, you can
     use line control to inform the compiler of where each source line
     originally came from.

   C preprocessors vary in some details. This manual discusses the Frexx C
preprocessor, which supports all by ANSI C specified syntaxes.

 Menu:

HTML Conversion by *[AG2HTML.pl](#)* V2.951201, perl 5.003 & *[witbrock@cs.cmu.edu](#)*

Global Actions

<pre>
                    Transformations Made Globally
                    =============================
</pre>

   Most C preprocessor features are inactive unless you give specific
directives to request their use. (Preprocessing directives are lines starting
with `#'). But there are three transformations that the preprocessor always
makes on all the input it receives, even in the absence of directives.

   * All C comments are replaced with single spaces.

   * Backslash-Newline sequences are deleted, no matter where. This
     feature allows you to break long lines for cosmetic purposes
     without changing their meaning.

   * Predefined macro names are replaced with their expansions.

   The first two transformations are done *before* nearly all other parsing and
before preprocessing directives are recognized. Thus, for example, you can
split a line cosmetically with Backslash-Newline anywhere.

<pre>
        /*
        / # /
        */ defi\
        ne FO\
        O 10\
        20
</pre>

is equivalent into `#define FOO 1020'. You can split even an escape sequence
with Backslash-Newline. For example, you can split `"foo\bar"' between the
`\' and the `b' to get

<pre>
        "foo\\
        bar"
</pre>

This behavior is unclean: in all other contexts, a Backslash can be inserted
in a string constant as an ordinary character by writing a double Backslash,
and this creates an exception. But the ANSI C standard requires it. (Strict
ANSI C does not allow Newlines in string constants, so they do not consider
this a problem.)

   But there are a few exceptions to all three transformations.

   * C comments and predefined macro names are not recognized inside a
     `#include' directive in which the file name is delimited with `<'
     and `>'.

```
   *  C comments and predefined macro names are never recognized within a
      character or string constant.  (Strictly speaking, this is the
      rule, not an exception, but it is worth noting here anyway.)
```

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Preprocessing Directives

```
                   Preprocessing Directives
                   ========================
```

  Most preprocessor features are active only if you use preprocessing
directives to request their use.

  Preprocessing directives are lines in your program that start with `#'. The
`#' is followed by an identifier that is the "directive name". For example,
`#define' is the directive that defines a macro. Whitespace is also allowed
before and after the `#'.

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

  Some directive names 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. [Simple Macros](#)

  A preprocessing directive cannot be more than one line in normal
circumstances. It may be split cosmetically with Backslash-Newline, but that
has no effect on its meaning. Comments containing Newlines can also divide
the directive into multiple lines, but the comments are changed to Spaces
before the directive is interpreted. The only way a significant Newline can
occur in a preprocessing directive is within a string constant or character
constant. Note that most C compilers that might be applied to the output from
the preprocessor do not accept string or character constants containing
Newlines.

  The `#' and the directive name cannot come from a macro expansion. For
example, if `foo' is defined as a macro expanding to `define', that does not
make `#foo' a valid preprocessing directive.

Simple Macros

```
                    Simple Macros
                    -------------
```

  A "simple macro" is a kind of abbreviation. It is a name which stands for a
fragment of code. Some people refer to these as "manifest constants".

  Before you can use a macro, you must "define" it explicitly with the
`#define' directive.  `#define' is followed by the name of the macro and then
the code it should be an abbreviation for. For example,

```
     #define BUFFER_SIZE 1020
```

defines a macro named `BUFFER_SIZE' as an abbreviation for the text `1020'. If
somewhere after this `#define' directive there comes a C statement of the
form

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

then the C preprocessor will recognize and "expand" the macro `BUFFER_SIZE',
resulting in

```
     foo = (char *) xmalloc (1020);
```

  The use of all upper case for macro names is a standard convention. Programs
are easier to read when it is possible to tell at a glance which names are
macros.

  Normally, a macro definition must be a single line, like all C preprocessing
directives. (You can split a long macro definition cosmetically with
Backslash-Newline.) There is one exception: Newlines can be included in the
macro definition if within a string or character constant. This is because it
is not possible for a macro definition to contain an unbalanced quote
character; the definition automatically extends to include the matching quote
character that ends the string or character constant. Comments within a macro
definition may contain Newlines, which make no difference since the comments
are entirely replaced with Spaces regardless of their contents.

  Aside from the above, there is no restriction on what can go in a macro
body. Parentheses need not balance. The body need not resemble valid C code.
(But if it does not, you may get error messages from the C compiler when you
use the macro.)

  The C preprocessor scans your program sequentially, so macro definitions
take effect at the place you write them. Therefore, the following input to
the C preprocessor

```
     foo = X;
     #define X 4
     bar = X;
```

produces as output

```
     foo = X;

     bar = 4;
```

   After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
     #define BUFSIZE 1020
     #define TABLESIZE BUFSIZE
```

the name `TABLESIZE' when used in the program would go through two stages of expansion, resulting ultimately in `1020'.

   This is not at all the same as defining `TABLESIZE' to be `1020'. The `#define' for `TABLESIZE' uses exactly the body you specify--in this case, `BUFSIZE'--and does not check to see whether it too is the name of a macro. It's only when you *use* `TABLESIZE' that the result of its expansion is checked for more macro names.

[Contents]

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Macros

```
                          Macros
                          ======
```

   A macro is a sort of abbreviation which you can define once and then use
later. There are many complicated features associated with macros in the C
preprocessor.

   Menu:

   [Simple Macros](#)   Macros that always expand the same way.
   [Argument Macros](#)   Macros that accept arguments that are substituted
                          into the macro expansion.
   [Predefined](#)   Predefined macros that are always available.
   [Stringification](#)   Macro arguments converted into string constants.
   [Concatenation](#)   Building tokens from parts taken from macro arguments.
   [Undefining](#)   Cancelling a macro's definition.
   [Redefining](#)   Changing a macro's definition.
   [Macro Pitfalls](#)   Macros can confuse the unwary. Here we explain
                          several common problems and strange features.

HTML Conversion by *[AG2HTML.pl](#)* V2.951201, perl 5.003 & *[witbrock@cs.cmu.edu](#)*

Macros with Arguments

```
                Macros with Arguments
                ---------------------
```

   A simple macro always stands for exactly the same text, each time it is
used. Macros can be more flexible when they accept "arguments". Arguments are
fragments of code that you supply each time the macro is used. These fragments
are included in the expansion of the macro according to the directions in the
macro definition. A macro that accepts arguments is called a "function-like
macro" because the syntax for using it looks like a function call.

   To define a macro that uses arguments, you write a `#define' directive with
a list of "argument names" in parentheses after the name of the macro. The
argument names may be any valid C identifiers, separated by commas and
optionally whitespace. The open-parenthesis must follow the macro name
immediately, with no space in between.

   For example, here is a macro that computes the minimum of two numeric
values, as it is defined in many C programs:

```
    #define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

    To use a macro that expects arguments, you write the name of the macro
followed by a list of "actual arguments" in parentheses, separated by commas.
The number of actual arguments you give must match the number of arguments the
macro expects. Examples of use of the macro `min' include `min (1, 2)' and
`min (x + 28, *p)'.

   The expansion text of the macro depends on the arguments you use. Each of
the argument names of the macro is replaced, throughout the macro definition,
with the corresponding actual argument. Using the same macro `min' defined
above, `min (1, 2)' expands into

```
    ((1) < (2) ? (1) : (2))
```

where `1' has been substituted for `X' and `2' for `Y'.

   Likewise, `min (x + 28, *p)' expands into

```
    ((x + 28) < (*p) ? (x + 28) : (*p))
```

   Parentheses in the actual arguments must balance; a comma within parentheses
does not end an argument. However, there is no requirement for brackets or
braces to balance, and they do not prevent a comma from separating arguments.
Thus,

```
    macro (array[x = y, x + 1])
```

passes two arguments to `macro': `array[x = y' and `x + 1]'. If you want to
supply `array[x = y, x + 1]' as an argument, you must write it as `array[(x =
y, x + 1)]', which is equivalent C code.

   After the actual arguments are substituted into the macro body, the entire
result is appended to the front of the remaining input, and the check for
macro calls continues. Therefore, the actual arguments can contain calls to
other macros, either with or without arguments, or even to the same macro. The
macro body can also contain calls to other macros. For example, `min (min (a,
b), c)' expands into this text:

```
    ((((a) < (b) ? (a) : (b))) < (c)
     ? (((a) < (b) ? (a) : (b)))
     : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

   If a macro `foo' takes one argument, and you want to supply an empty
argument, you must write at least some whitespace between the parentheses,
like this: `foo ( )'.  Just `foo ()' is providing no arguments, which is an
error if `foo' expects an argument.  But `foo0 ()' is the correct way to call
a macro defined to take zero arguments, like this:

```
    #define foo0() ...
```

   If you use the macro name followed by something other than an
open-parenthesis (after ignoring any spaces, tabs and comments that follow),
it is not a call to the macro, and the preprocessor does not change what you
have written. Therefore, it is possible for the same name to be a variable or
function in your program as well as a macro, and you can choose in each
instance whether to refer to the macro (if an actual argument list follows) or
the variable or function (if an argument list does not follow).

   Such dual use of one name could be confusing and should be avoided except
when the two meanings are effectively synonymous: that is, when the name is
both a macro and a function and the two have similar effects. You can think
of the name simply as a function; use of the name for purposes other than
calling it (such as, to take the address) will refer to the function, while
calls will expand the macro and generate better but equivalent code. For
example, you can use a function named `min' in the same source file that
defines the macro. If you write `&min' with no argument list, you refer to the
function. If you write `min (x, bb)', with an argument list, the macro is
expanded. If you write `(min) (a, bb)', where the name `min' is not followed
by an open-parenthesis, the macro is not expanded, so you wind up with a call
to the function `min'.

   You may not define the same name as both a simple macro and a macro with
arguments.

   In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

      #define FOO(x) - 1 / (x)

(which defines `FOO' to take an argument and expand into minus the reciprocal of that argument) or this:

      #define BAR (x) - 1 / (x)

(which defines `BAR' to take no argument and always expand into `(x) - 1 / (x)').

   Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

HTML Conversion by AG2HTML.pl V2.951201, perl 5.003 & witbrock@cs.cmu.edu

Predefined Macros

```
                    Predefined Macros
                    -----------------
```

   Several simple macros are predefined. You can use them without giving
definitions for them. They fall into two classes: standard macros and
system-specific macros.

 Menu:

[Standard Predefined](#)   Standard predefined macros.
[Nonstandard Predefined](#)   Nonstandard predefined macros.

Standard Predefined Macros

```
                    Standard Predefined Macros
                    ..........................
```

   The standard predefined macros are available with the same meanings
regardless of the machine or operating system on which you are using the Frexx
cpp. Their names all start and end with double underscores. The first four
defines in this table are standardized by ANSI C. The rest are Frexx
extensions.

`__FILE__'
     This macro expands to the name of the current input file, in the
     form of a C string constant. The precise name returned is the one
     that was specified in `#include' or as the input file name
     argument.

`__LINE__'
     This macro expands to the current input line number, in the form
     of a decimal integer constant. While we call it a predefined
     macro, it's a pretty strange macro, since its "definition" changes
     with each new line of source code.

     This and `__FILE__' are useful in generating an error message to
     report an inconsistency detected by the program; the message can
     state the source line at which the inconsistency was detected.
     For example,

```
          fprintf (stderr, "Internal error: "
                           "negative string length "
                           "%d at %s, line %d.",
                   length, __FILE__, __LINE__);
```

     A `#include' directive changes the expansions of `__FILE__' and
     `__LINE__' to correspond to the included file. At the end of that
     file, when processing resumes on the input file that contained the
     `#include' directive, the expansions of `__FILE__' and `__LINE__'
     revert to the values they had before the `#include' (but
     `__LINE__' is then incremented by one as processing moves to the
     line after the `#include').

     The expansions of both `__FILE__' and `__LINE__' are altered if a
     `#line' directive is used. [Combining Sources](#)

`__DATE__'
     This macro expands to a string constant that describes the date on
     which the preprocessor is being run. The string constant contains
     eleven characters and looks like `"Jan 29 1987"' or  `"Apr 1 1905"'.

`__TIME__'

     This macro expands to a string constant that describes the time at
     which the preprocessor is being run. The string constant contains
     eight characters and looks like `"23:59:01"'.

'__FUNCTION__'

     This macro expands to a name of the currently processed function, in
     the form of a C string constant.

'__FUNC_LINE__'

     This macro expands to the current function's starting line number, in
     the form of a decimal integer constant.

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Nonstandard Predefined Macros

```
                Nonstandard Predefined Macros
                .............................
```

   The C preprocessor normally has several predefined macros that vary between
machines because their purpose is to indicate what type of system and machine
is in use. This manual, being for all systems and machines, cannot tell you
exactly what their names are; instead, we offer a list of some typical ones.

   Some nonstandard predefined macros describe the operating system in use,
with more or less specificity. The Frexx cpp has not been widely developed in
this area, why only a few symbols are offered:

`unix'
      `unix' is normally predefined on all Unix systems.

'amiga' 'amigados'
      'amiga' and 'amigados' is normally predefined on all Amiga systems.

`m68000'
      `m68000' is predefined on most computers whose CPU is a Motorola
      680x0.

   These predefined symbols are not only nonstandard, they are contrary to the
ANSI standard because their names do not start with underscores.

Macro Stringification

```
                        Stringification
                        ---------------
```

   "Stringification" means turning a code fragment into a string constant whose
contents are the text for the code fragment. For example, stringifying `foo
(z)' results in `"foo (z)"'.

   In the C preprocessor, stringification is an option available when macro
arguments are substituted into the macro definition. In the body of the
definition, when an argument name appears, the character `#' before the name
specifies stringification of the corresponding actual argument when it is
substituted at that point in the definition. The same argument may be
substituted in other places in the definition without stringification if the
argument name appears in those places with no `#'.

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

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

Here the actual argument for `EXP' is substituted once as given, into the `if'
statement, and once as stringified, into the argument to `fprintf'.  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 [Swallow Semicolon](#).

   The stringification feature is limited to transforming one macro argument
into one string constant: there is no way to combine the argument with other
text and then stringify it all together. But the example above shows how an
equivalent result can be obtained in ANSI Standard C using the feature that
adjacent string constants are concatenated as one string constant. The
preprocessor stringifies the actual value of `EXP' into a separate string
constant, resulting in text like

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

but the C compiler then sees three consecutive string constants and
concatenates them into one, producing effectively

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

Stringification in C involves more than putting doublequote characters around the fragment; it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying `p = "foo\n";' results in `"p = \"foo\\n\";"'. However, backslashes that are not inside of string or character constants are not duplicated: `\n' by itself stringifies to `"\n"'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

Swallowing the Semicolon

```
                Swallowing the Semicolon
                .......................
```

   Often it is desirable to define a macro that expands into a compound
statement. Consider, for example, the following macro, that advances a
pointer (the argument `p' says where to find it) across whitespace
characters:

```
    #define SKIP_SPACES (p, limit)  \
    { register char *lim = (limit); \
      while (p != lim) {             \
        if (*p++ != ' ') {          \
          p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a
single line, so that it resembles the way such C code would be laid out if not
part of a macro definition.

   A call to this macro might be `SKIP_SPACES (p, lim)'. Strictly speaking,
the call expands to a compound statement, which is a complete statement with
no need for a semicolon to end it. But it looks like a function call. So it
minimizes confusion if you can use it like a function call, writing a
semicolon afterward, as in `SKIP_SPACES (p, lim);'

   But this can cause trouble before `else' statements, because the semicolon
is actually a null statement. Suppose you write

```
    if (*p != 0)
      SKIP_SPACES (p, lim);
    else ...
```

The presence of two statements--the compound statement and a null
statement--in between the `if' condition and the `else' makes invalid C code.

   The definition of the macro `SKIP_SPACES' can be altered to solve this
problem, using a `do ... while' statement. Here is how:

```
    #define SKIP_SPACES (p, limit)     \
    do { register char *lim = (limit); \
         while (p != lim) {            \
           if (*p++ != ' ') {          \
             p--; break; }}}           \
    while (0)
```

   Now `SKIP_SPACES (p, lim);' expands into

```
    do {...} while (0);
```

which is one statement.

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Macro Concatenation

```
                    Concatenation
                    -------------
```

   "Concatenation" means joining two strings into one. In the context of macro
expansion, concatenation refers to joining two lexical units into one longer
one. Specifically, an actual argument to the macro can be concatenated with
another actual argument or with fixed text to produce a longer name. The
longer name might be the name of a function, variable or type, or a C keyword;
it might even be the name of another macro, in which case it will be
expanded.

   When you define a macro, you request concatenation with the special operator
`##' in the macro body. When the macro is called, after actual arguments are
substituted, all `##' operators are deleted, and so is any whitespace next to
them (including whitespace that was part of an actual argument). The result
is to concatenate the syntactic tokens on either side of the `##'.

   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) ();
    };

    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 stringification, 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),
```

```
        ...
      };
```

  The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But 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 concatenation.  In some cases it is even possible to piece together a string constant.  However, two pieces of text that don't together form a valid lexical unit cannot be concatenated.  For example, concatenation with `x' on one side and `+' on the other is not meaningful because those two characters can't fit together in any lexical unit of C.  The ANSI standard says that such attempts at concatenation are undefined, but in the Frexx C preprocessor it is well defined: it puts the `x' and `+' side by side with no particular special results.

  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 `*': the `/*' sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character.  Also, you can freely use comments next to a `##' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

NOTE:

  The order of concatenation/macro expantion is unspecified in the ANSI standard. If the right parts itself is a defined string, it is not known wheather the defined string will be expanded before the concatenation or if the concatenation is done first.
  Most ANSI compilers first append the "real" words and so does the Frexx C preprocessor as long as -R isn't specified. If -R is specified, the right part of the concat is first subject to substitution, and then append is done, to still be compatible with certain sources.
  Example:

```
        #define FOOBAR fooBAR
        #define append(x,y) x ## y
        #define BAR bar

        append(FOO, BAR)

        Result without -R: fooBAR
        Result with    -R: FOObar
```

Undefining Macros

```
                    Undefining Macros
                    -----------------
```

  To "undefine" a macro means to cancel its definition. This is done with the
`#undef' directive. `#undef' is followed by the macro name to be undefined.

  Like definition, undefinition occurs at a specific point in the source file,
and it applies starting from that point. The name ceases to be a macro name,
and from that point on it is treated by the preprocessor as if it had never
been a macro name.

    For example,

```
    #define FOO 4
    x = FOO;
    #undef FOO
    x = FOO;
```

expands into

```
    x = 4;

    x = FOO;
```

In this example, `FOO' had better be a variable or function as well as
(temporarily) a macro, in order for the result of the expansion to be valid C
code.

  The same form of `#undef' directive will cancel definitions with arguments
or definitions that don't expect arguments. The `#undef' directive has no
effect when used on a name not currently defined as a macro.

Redefining Macros

```
             Redefining Macros
             -----------------
```

   "Redefining" a macro means defining (with `#define') a name that is already
defined as a macro.

   A redefinition is trivial if the new definition is transparently identical
to the old one. You probably wouldn't deliberately write a trivial
redefinition, but they can happen automatically when a header file is included
more than once, so they are accepted silently and without effect.

   Nontrivial redefinition is considered likely to be an error, so it provokes
a warning message from the preprocessor. However, sometimes it is useful to
change the definition of a macro in mid-compilation. You can inhibit the
warning by undefining the macro with `#undef' before the second definition.

   In order for a redefinition to be trivial, the new definition must exactly
match the one already in effect, with two possible exceptions:

   * Whitespace may be added or deleted at the beginning or the end.

   * Whitespace may be changed in the middle (but not inside strings).
     However, it may not be eliminated entirely, and it may not be added
     where there was no whitespace at all.

   Recall that a comment counts as whitespace.

Pitfalls and Subtleties of Macros

     In this section we describe some special rules that apply to macros and
macro expansion, and point out certain cases in which the rules have
counterintuitive consequences that you must watch out for.

 Menu

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Header Files

```
                    Header Files
                    ============
```

   A header file is a file containing C declarations and macro definitions to
be shared between several source files. You request the use of a header file
in your program with the C preprocessing directive `#include'.

 Menu:

[Header Uses](.)   What header files are used for.
[Include Syntax](.)   How to write `#include' directives.
[Include Operation](.)   What `#include' does.
[Once-Only](.)   Preventing multiple inclusion of one header file.
[Inheritance](.)   Including one header file in another header file.

HTML Conversion by *[AG2HTML.pl](.)* V2.951201, perl 5.003 & *[witbrock@cs.cmu.edu](.)*

Uses of Header Files

```
                    Uses of Header Files
                    --------------------
```

Header files serve two kinds of 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 in C compilation as
copying the header file into each source file that needs it. But 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.

The usual convention is to give header files names that end with `.h'.
Avoid unusual characters in header file names, as they reduce portability.

[Contents]

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

The '#include' Directive

```
                    The `#include' Directive
                    ------------------------
```

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

`#include <FILE>'
     This variant is used for system header files. It searches for a
     file named FILE in a list of directories specified by you, then in
     a standard list of system directories. You specify directories to
     search for header files with the command option `-I' (Invocation).

     The parsing of this form of `#include' is slightly special because
     comments are not recognized within the `<...>'. Thus, in
     `#include <x/*y>' the `/*' does not start a comment and the
     directive specifies inclusion of a system header file named
     `x/*y'. Of course, a header file with such a name is unlikely to
     exist on Unix, where shell wildcard features would make it hard to
     manipulate.

     The argument FILE may not contain a `>' character. It may,
     however, contain a `<' character.

`#include "FILE"'
     This variant is used for header files of your own program. It
     searches for a file named FILE first in the current directory,
     then in the same directories used for system header files. The
     current directory is the directory of the current input file. It
     is tried first because it is presumed to be the location of the
     files that the current input file refers to. (If the `-I-' option
     is used, the special treatment of the current directory is
     inhibited.)

     The argument FILE may not contain `"' characters. 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. It is not clear why this behavior is ever useful, but
     the ANSI standard specifies it.

`#include ANYTHING ELSE'
     This variant is called a "computed #include". Any `#include'

directive whose argument does not fit the above two forms is a
computed include. The text ANYTHING ELSE is checked for macro
calls, which are expanded. When this is done, the result must fit
one of the above two variants--in particular, the expanded text
must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file
name to be used at a later point in the program. One application
of this is to allow a site-specific configuration file for your
program to specify the names of the system include files to be
used. This can help in porting the program to various operating
systems in which the necessary system header files are found in
different places.

[Contents]

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Invoking the C Preprocessor

```
                 Invoking the C Preprocessor
                 ===========================
```

   Most often when you use the C preprocessor you will not have to invoke it
explicitly: the C compiler will do so automatically. However, the preprocessor
is sometimes useful on its own.

   The C preprocessor expects two file names as arguments, INFILE and OUTFILE.
The preprocessor reads INFILE together with any other files it specifies with
`#include'. All the output generated by the combined input files is written
in OUTFILE.

   Either INFILE or OUTFILE may be `-', which as INFILE means to read from
standard input and as OUTFILE means to write to standard output. Also, if
OUTFILE or both file names are omitted, the standard output and standard input
are used for the omitted file names.

   These options are recognized by the Frexx C preprocessor:

   -B CPP normally predefines some symbols defining the target computer
       and operating system. If -B is specified, no such symbols will be
       predefined.

   -b Warnings will be displayed if there isn't as many open as close
       characters of the parentheses, brackets and braces symbols.

   -C If set, source-file comments are written to the output. This
       allows the output of CPP to be used as the input to a program,
       such as lint, that expects commands embedded in specially-
       formatted comments.

   -Dname=value   Define the name as if the programmer wrote

                     #define name value

                   at the start of the first file. If "=value" is not
                   given, a value of "1" will be used.

   -d Display all given options, including input and output files.

   -E Always return "success" to the operating system, even if errors
       were detected. Note that some fatal errors will terminate CPP,
       returning "failure" even if the -E option is given.

   -F Print the pathnames of included files, one per line on the

       standard error.

-H Try to keep all whitespaces as found in the source. This is useful
    when you want an output using the same indent as the source has.
    Otherwise, any number of whitespaces will be replaced with one
    single space (' ').

-h Output help text.

-Idirectory   Add   this directory to the list   of directories
                 searched   for #include "..."   and #include <...>
                 commands. Note that there is no space between the "-I"
                 and the directory string (that must end with a slash
                 '/'). More than one -I command is permitted.

-J Allow nested comments.

-j Warn whenever a nested comment is discovered.

-LL Preprocesses input without producing line control information for
    the next pass of the C compiler. This also produces an output
    without unnecessary empty lines.

-L Output "# <line>" prior to "#line <line>".

-M Disable warnings when an include file isn't found.

-N If this is specified, the "always present" symbols, __LINE__,
    __FILE__, __TIME__, __DATE__, __FUNCTION__ and __FUNC_LINE__ are
    not defined.

-P Do not recognize and remove C++ style comments.

-p Enable warnings on non ANSI preprocessor instructions. When this
    option is enabled, all #-keywords that are not specified in the
    ANSI standard X3J11 will be reported with warnings.

-Q Makes cpp ignore and visualize all unrecognized flags. This flag
    was implemented to make it possible to use my cpp with the default
    AIX 'cc' compiler. Since   that compiler always calls   the
    preprocessor with some other flags not identified by this
    compiler, I had to do this...

-q Same as -Q, but silent. Nothing is output when unknown options are
    ignored.

-R In situations where concatenated macros are used like:
    #define FOOBAR fooBAR
    #define append(x,y) x ## y
    #define BAR bar

```
      append(FOO, BAR)

      Result without -R: fooBAR
      Result with    -R: FOObar
```

It is unspecified in the ANSI draft in which order to evaluate
this. Should the "real" second word first be appended before the
macro substitution occurs, or should the word get subsitituted
first? Most ANSI compilers first append the "real" words and so
does 'cpp' if -R isn't specified. If -R is specified, the right
part of the concat is first subject to substitution, and then
append is done.

-Stext   cpp normally assumes that thesize of the target computer's
         basic variable types is the same as the size of these
         types of the host computer. The -S option allows dynamic
         respecification of these values. "text" is a string of
         numbers, separated by commas, that specifies correct sizes.
         The sizes must be specified in the exact order:

              char short int long float double

         If you specify the option as "-S*text", pointers to these
         types will be specified. -S* takes one additional argument
         for pointer to function (e.g. int (*)())

         For example, to specify sizes appropriate for a PDP-11, you
         would write:

                c s i l f d func
              -S1,2,2,2,4,8,
              -S*2,2,2,2,2,2,2

         Note that *ALL* values must be specified.

-Uname   Undefine the name as if

              #undef name

         were given.

-V Do not output the version information at startup.

-W Outputs all #defines at the end of the output file.

-w Only output #defines and nothing else.

-X #Includes the specified file at the top of the source file.

How '#include' works

```
                   How `#include' Works
                   --------------------
```

   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,
given a header file `header.h' as follows,

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

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

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

    main ()
    {
      printf (test ());
    }
```

the output generated by the C preprocessor for `program.c' as input would be

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

    main ()
    {
      printf (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 could even contain the beginning of a statement
that is concluded in the containing file, or the end of a statement that was
started in the including file. However, a comment or a string or character
constant may not start in the included file and finish in the including file.
An unterminated comment, string constant or character constant in an included
file is considered to end (with an error message) at the end of the file.

   It is possible for a header file to begin or end a syntactic unit such as a
function definition, but that would be very confusing, so don't do it.

   The line following the `#include' directive is always treated as a separate
line by the C preprocessor even if the included file lacks a final newline.

Once-Only Include Files

```
                    Once-Only Include Files
                    -----------------------
```

   Very often, one header file includes another. It can easily result that a
certain header file is included more than once. This may lead to errors, if
the header file defines structure types or typedefs, and is certainly
wasteful. Therefore, we often wish to prevent multiple inclusion of a header
file.

   The standard way to do this is to enclose the entire real contents of the
file in a conditional, like this:

```
    #ifndef FILE_FOO_SEEN
    #define FILE_FOO_SEEN

    THE ENTIRE FILE

    #endif /* FILE_FOO_SEEN */
```

   The macro `FILE_FOO_SEEN' indicates that the file has been included once
already. In a user header file, the macro name should not begin with `_'. In
a system header file, this name 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.

[Contents]

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Inheritance and Header Files

```
                    Inheritance and Header Files
                    ----------------------------
```

   "Inheritance" is what happens when one object or file derives some of its
contents by virtual copying from another object or file. In the case of C
header files, inheritance means that one header file includes another header
file and then replaces or adds something.

   If the inheriting header file and the base header file have different names,
then inheritance is straightforward: simply write `#include "BASE"' in the
inheriting file.

   Sometimes it is necessary to give the inheriting file the same name as the
base file. This is less straightforward.

   For example, suppose an application program uses the system header file
`sys/signal.h', but the version of `/usr/include/sys/signal.h' on a particular
system doesn't do what the application program expects. It might be convenient
to define a "local" version, perhaps under the name
`/usr/local/include/sys/signal.h', to override or add to the one supplied by
the system.

   You can do this by using the option `-I.' for compilation, and writing a
file `sys/signal.h' that does what the application program expects. But
making this file include the standard `sys/signal.h' is not so easy--writing
`#include <sys/signal.h>' in that file doesn't work, because it includes your
own version of the file, not the standard system version. Used in that file
itself, this leads to an infinite recursion and a fatal error in compilation.

   `#include </usr/include/sys/signal.h>' would find the proper file, but that
is not clean, since it makes an assumption about where the system header file
is found. This is bad for maintenance, since it means that any change in
where the system's header files are kept requires a change somewhere else.

   The clean way to solve this problem is to use `#include_next', which means,
"Include the *next* file with this name." This directive works like
`#include' except in searching for the specified file: it starts searching the
list of header file directories *after* the directory in which the current
file was found.

   Suppose you specify `-I /usr/local/include', and the list of directories to
search also includes `/usr/include'; and suppose that both directories contain
a file named `sys/signal.h'.  Ordinary `#include <sys/signal.h>' finds the
file under `/usr/local/include'. If that file contains `#include_next
<sys/signal.h>', it starts searching after that directory, and finds the file
in `/usr/include'.

Combining Source Files

```
            Combining Source Files
            =======================
```

   One of the jobs of the C preprocessor is to inform the C compiler of where
each line of C code came from: which source file and which line number.

   C code can come from multiple source files if you use `#include'; both
`#include' and the use of conditionals and macros can cause the line number of
a line in the preprocessor output to be different from the line's number in
the original source file. You will appreciate the value of making both the C
compiler (in error messages) and symbolic debuggers such as GDB use the line
numbers in your source file.

   The C preprocessor builds on this feature by offering a directive by which
you can control the feature explicitly. This is useful when a file for input
to the C preprocessor is the output from another program such as the `bison'
parser generator, which operates on another file that is the true source file.
Parts of the output from `bison' are generated from scratch, other parts come
from a standard parser file. The rest are copied nearly verbatim from the
source file, but their line numbers in the `bison' output are not the same as
their original line numbers. Naturally you would like compiler error messages
and symbolic debuggers to know the original source file and line number of
each line in the `bison' input.

   `bison' arranges this by writing `#line' directives into the output file.
`#line' is a directive that specifies the original line number and source file
name for subsequent input in the current preprocessor input file. `#line' has
three variants:

`#line LINENUM'
     Here LINENUM is a decimal integer constant. This specifies that
     the line number of the following line of input, in its original
     source file, was LINENUM.

`#line LINENUM FILENAME'
     Here LINENUM is a decimal integer constant and FILENAME is a
     string constant. This specifies that the following line of input
     came originally from source file FILENAME and its line number there
     was LINENUM. Keep in mind that FILENAME is not just a file name;
     it is surrounded by doublequote characters so that it looks like a
     string constant.

`#line ANYTHING ELSE'
     ANYTHING ELSE is checked for macro calls, which are expanded. The
     result should be a decimal integer constant followed optionally by
     a string constant, as described above.

   `#line' directives alter the results of the `__FILE__' and `__LINE__'
predefined macros from that point on. SEE 'Standard Predefined Symbols'.

   The output of the preprocessor (which is the input for the rest of the
compiler) contains directives that look much like `#line' directives. They
start with just `#' instead of `#line', but this is followed by a line number
and file name as in `#line'.

HTML Conversion by *[AG2HTML.pl](#)* V2.951201, perl 5.003 & *[witbrock@cs.cmu.edu](#)*

Conditionals

```
                 Conditionals
                 ============
```

   In a macro processor, a "conditional" is a directive that allows a part of
the program to be ignored during compilation, on some conditions. In the C
preprocessor, a conditional can test either an arithmetic expression or
whether a name is defined as a macro.

   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.

[Uses](#)   What conditionals are for.
[Syntax](#)   How conditionals are written.
[Deletion](#)   Making code into a comment.
[Macros](#)   Why conditionals are used with macros.
[Assertions](#)   How and why to use assertions.
[Errors](#)   (#error Directive) Detecting inconsistent compilation parameters.

Why Conditionals are Used

```
                Why Conditionals are Used
                -------------------------
```

 Generally there are three kinds of reason to use a conditional.

 * A program may need to use different code depending on the machine
   or operating system it is to run on. In some cases the code for
   one operating system may be erroneous on another operating system;
   for example, it might refer to library routines that do not exist
   on the other system. When this happens, it is not enough to avoid
   executing the invalid code: merely having it in the program makes
   it impossible to link the program and run it. With a
   preprocessing conditional, the offending code can be effectively
   excised from the program when it is not valid.

 * You may want to be able to compile the same source file into two
   different programs. Sometimes the difference between the programs
   is that one makes frequent time-consuming consistency checks on its
   intermediate data, or prints the values of those data for
   debugging, while the other does not.

 * A conditional whose condition is always false is a good way to
   exclude code from the program but keep it as a sort of comment for
   future reference.

   Most simple programs that are intended to run on only one machine will not
need to use preprocessing conditionals.

Syntax of Conditionals

```
                 Syntax of Conditionals
                 ----------------------


  A conditional in the C preprocessor begins with a "conditional directive":
`#if', `#ifdef' or `#ifndef'. See Conditionals-Macros, for information on
`#ifdef' and `#ifndef'; only `#if' is explained here.

 Menu:

 If: #if Directive.      Basic conditionals using `#if' and `#endif'.
 Else: #else Directive.  Including some text if the condition fails.
 Elif: #elif Directive.  Testing several alternative possibilities.
```

Keeping Deleted Code for Future Reference

```
            Keeping Deleted Code for Future Reference
            -------------------------------------------
```

   If you replace or delete a part of the program but want to keep the old code
around as a comment for future reference, the easy way to do this is to put
`#if 0' before it and `#endif' after it. This is better than using comment
delimiters `/*' and `*/' since those won't work if the code already contains
comments (C comments do not nest).

   This works even if the code being turned off contains conditionals, but they
must be entire conditionals (balanced `#if' and `#endif').

   Conversely, do not use `#if 0' for comments which are not C code. Use the
comment delimiters `/*' and `*/' instead. The interior of `#if 0' must consist
of complete tokens; in particular, singlequote characters must balance. But
comments often contain unbalanced singlequote characters (known in English as
apostrophes). These confuse `#if 0'.  They do not confuse `/*'.

Conditionals and Macros

```
          Conditionals and Macros
          -----------------------
```

   Conditionals are useful in connection with macros or assertions, because
those are the only ways that an expression's value can vary from one
compilation to another. A `#if' directive whose expression uses no macros or
assertions is equivalent to `#if 1' or `#if 0'; you might as well determine
which one, by computing the value of the expression yourself, and then
simplify the program.

   For example, here is a conditional that tests the expression `BUFSIZE ==
1020', where `BUFSIZE' must be a macro.

```
    #if BUFSIZE == 1020
      printf ("Large buffers!\n");
    #endif /* BUFSIZE is large */
```

   (Programmers often wish they could test the size of a variable or data type
in `#if', but even if this isn't ANSI specified behaviour, the Frexx C
preprocessor understands `sizeof'.)

   The special operator `defined' is used in `#if' expressions to test whether
a certain name is defined as a macro. Either `defined NAME' or `defined
(NAME)' is an expression whose value is 1 if NAME is defined as macro at the
current point in the program, and 0 otherwise. For the `defined' operator it
makes no difference what the definition of the macro is; all that matters is
whether there is a definition. Thus, for example,

```
    #if defined (vax) || defined (ns16000)
```

would succeed if either of the names `vax' and `ns16000' is defined as a
macro. You can test the same condition using assertions ([Assertions](#)), like
this:

```
    #if #cpu (vax) || #cpu (ns16000)
```

   If a macro is defined and later undefined with `#undef', subsequent use of
the `defined' operator returns 0, because the name is no longer defined. If
the macro is defined again with another `#define', `defined' will recommence
returning 1.

   Conditionals that test whether just one name is defined are very common, so
there are two special short conditional directives for this case.

`#ifdef NAME'

```
     is equivalent to `#if defined (NAME)'.
```

`#ifndef NAME'
```
     is equivalent to `#if ! defined (NAME)'.
```

 Macro definitions can vary between compilations for several reasons.

 * Some macros are predefined on each kind of machine. For example,
   on a Vax, the name `vax' is a predefined macro. On other
   machines, it would not be defined.

 * Many more macros are defined by system header files. Different
   systems and machines define different macros, or give them
   different values. It is useful to test these macros with
   conditionals to avoid using a system feature on a machine where it
   is not implemented.

 * Macros are a common way of allowing users to customize a program
   for different machines or applications. For example, the macro
   `BUFSIZE' might be defined in a configuration file for your
   program that is included as a header file in each source file. You
   would use `BUFSIZE' in a preprocessing conditional in order to
   generate different code depending on the chosen configuration.

 * Macros can be defined or undefined with `-D' and `-U' command
   options when you compile the program. You can arrange to compile
   the same source file into two different programs by choosing a
   macro name to specify which program you want, writing conditionals
   to test whether or how this macro is defined, and then controlling
   the state of the macro with compiler command options. [Invocation](#)

   Assertions are usually predefined, but can be defined with preprocessor
directives or command-line options.

Assertions

```
                    Assertions
                    ----------
```

Not implemented in the Frexx C preprocessor!

HTML Conversion by *[AG2HTML.pl](#)* V2.951201, perl 5.003 & *[witbrock@cs.cmu.edu](#)*

The #error and #warning Directives

```
          The `#error' and `#warning' Directives
          ---------------------------------------
```

   The directive `#error' causes the preprocessor to report a fatal error.  The rest of the line that follows `#error' is used as the error message.

   You would use `#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
     #ifdef __vax__
     #error Won't work on Vaxen.  See comments at get_last_object.
     #endif
```

See 'Nonstandard Predefined Symbols', for why this works.

   If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with `#error'. For example,

```
     #if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
         || HASH_TABLE_SIZE % 5 == 0
     #error HASH_TABLE_SIZE should not be divisible by a small prime
     #endif
```

   The directive `#warning' is like the directive `#error', but causes the preprocessor to issue a warning and continue preprocessing. The rest of the line that follows `#warning' is used as the warning message.

   You might use `#warning' in obsolete header files, with a message directing the user to the header file which should be used instead.

Miscellaneous Preprocessing Directives

```
          Miscellaneous Preprocessing Directives
          ======================================
```

   This section describes three additional preprocessing directives. They are
not very useful, but are mentioned for completeness.

   The "null directive" consists of a `#' followed by a Newline, with only
whitespace (including comments) in between. A null directive is understood as
a preprocessing directive but has no effect on the preprocessor output. The
primary significance of the existence of the null directive is that an input
line consisting of just a `#' will produce no output, rather than a line of
output containing just a `#'. Supposedly some old C programs contain such
lines.

   The ANSI standard specifies that the `#pragma' directive has an arbitrary,
implementation-defined effect. In the Frexx C preprocessor, `#pragma'
directives are not used. However, they are left in the preprocessor output, so
they are available to the compilation pass.

   The `#ident' directive is supported for compatibility with certain other
systems. It is followed by a line of text. On some systems, the text is
copied into a special place in the object file; on most systems, the text is
ignored and this directive has no effect. Typically `#ident' is only used in
header files supplied with those systems where it is meaningful.

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

C Preprocessor Output

```
             C Preprocessor Output
             ======================
```

   The output from the C preprocessor looks much like the input, except that
all preprocessing directive lines have been replaced with blank lines and all
comments with spaces. Whitespace within a line is not altered; however, a
space is inserted after the expansions of most macro calls.

   Source file name and line number information is conveyed by lines of the
form

```
     # LINENUM FILENAME
```

which are inserted as needed into the middle of the input (but never within a
string or character constant). Such a line means that the following line
originated in file FILENAME at line LINENUM.

Index

@remark Index auto-generated by Heddley
Index of database 076adec8-0

Documents

Buttons

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Cascaded Use of Macros

```
               Cascaded Use of Macros
               ......................


   A "cascade" of macros is when one macro's body contains a reference to
another macro.  This is very common practice.  For example,

     #define BUFSIZE 1020
     #define TABLESIZE BUFSIZE


   This is not at all the same as defining `TABLESIZE' to be `1020'. The
`#define' for `TABLESIZE' uses exactly the body you specify--in this case,
`BUFSIZE'--and does not check to see whether it too is the name of a macro.

   It's only when you *use* `TABLESIZE' that the result of its expansion is
checked for more macro names.

   This makes a difference if you change the definition of `BUFSIZE' at some
point in the source file.  `TABLESIZE', defined as shown, will always expand
using the definition of `BUFSIZE' that is currently in effect:

     #define BUFSIZE 1020
     #define TABLESIZE BUFSIZE
     #undef BUFSIZE
     #define BUFSIZE 37

Now `TABLESIZE' expands (in two stages) to `37'.  (The `#undef' is to prevent
any warning about the nontrivial redefinition of `BUFSIZE'.)
```

Duplication of Side Effects

```
                 Duplication of Side Effects
                 ...........................
```

  Many C programs define a macro `min', for "minimum", like this:

```
    #define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

   When you use this macro with an argument containing a side effect, as shown
here,

```
    next = min (x + y, foo (z));
```

it expands as follows:

```
    next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where `x + y' has been substituted for `X' and `foo (z)' for `Y'.

   The function `foo' is used only once in the statement as it appears in the
program, but the expression `foo (z)' has been substituted twice into the
macro expansion. As a result, `foo' might be called two times when the
statement is executed. If it has side effects or if it takes a long time to
compute, the results might not be what you intended. We say that `min' is an
"unsafe" macro.

   The only solution is to be careful when *using* the macro `min'. For
example, you can calculate the value of `foo (z)', save it in a variable, and
use that variable in `min':

```
    #define min(X, Y)  ((X) < (Y) ? (X) : (Y))
    ...
    {
      int tem = foo (z);
      next = min (x + y, tem);
    }
```

(where we assume that `foo' returns type `int').

Improperly Nested Constructs

```
             Improperly Nested Constructs
             ............................
```

   Recall that when a macro is called with arguments, the arguments are
substituted into the macro body and the result is checked, together with the
rest of the input file, for more macro calls.

   It is possible to piece together a macro call coming partially from the
macro body and partially from the actual arguments. For example,

```
     #define double(x) (2*(x))
     #define call_with_1(x) x(1)
```

would expand `call_with_1 (double)' into `(2*(1))'.

   Macro definitions do not have to have balanced parentheses. By writing an
unbalanced open parenthesis in a macro body, it is possible to create a macro
call that begins inside the macro body but ends outside of it. For example,

```
     #define strange(file) fprintf (file, "%s %d",
     ...
     strange(stderr) p, 35)
```

This bizarre example expands to `fprintf (stderr, "%s %d", p, 35)'!

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

Newlines in Macro Arguments

```
             Newlines in Macro Arguments
             ---------------------------
```

   Traditional macro processing carries forward all newlines in macro arguments
into the expansion of the macro. This means that, if some of the arguments are
substituted more than once, or not at all, or out of order, newlines can be
duplicated, lost, or moved around within the expansion. If the expansion
consists of multiple statements, then the effect is to distort the line
numbers of some of these statements. The result can be incorrect line
numbers, in error messages or displayed in a debugger.

   Here is an example illustrating this problem:

```
     #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 four, even though the statement text comes from line five.

Self-Referential Macros

                    Self-Referential Macros
                    ....

   A "self-referential" macro is one whose name appears in its definition. A
special feature of ANSI Standard C is that the self-reference is not
considered a macro call. It is passed into the preprocessor output
unchanged.

   Let's consider an example:

      #define foo (4 + foo)

where `foo' is also a variable in your program.

   Following the ordinary rules, each reference to `foo' will expand into `(4 +
foo)'; then this will be rescanned and will expand into `(4 + (4 + foo))'; and
so on until it causes a fatal error (memory full) in the preprocessor.

   However, the special rule about self-reference cuts this process short after
one step, at `(4 + foo)'. Therefore, this macro definition has the possibly
useful effect of causing the program to add 4 to the value of `foo' wherever
`foo' is referred to.

   In most cases, it is a bad idea to take advantage of this feature. A person
reading the program who sees that `foo' is a variable will not expect that it
is a macro as well. The reader will come across the identifier `foo' in the
program and think its value should be that of the variable `foo', whereas in
fact the value is four greater.

   The special rule for self-reference applies also to "indirect"
self-reference. This is the case where a macro X expands to use a macro `y',
and the expansion of `y' refers to the macro `x'. The resulting reference to
`x' comes indirectly from the expansion of `x', so it is a self-reference and
is not further expanded. Thus, after

      #define x (4 + y)
      #define y (2 * x)

`x' would expand into `(4 + (2 * x))'. Clear?

   But suppose `y' is used elsewhere, not from the definition of `x'. Then the
use of `x' in the expansion of `y' is not a self-reference because `x' is not
"in progress". So it does expand. However, the expansion of `x' contains a
reference to `y', and that is an indirect self-reference now because `y' is
"in progress". The result is that `y' expands to `(2 * (4 + y))'.

   It is not clear that this behavior would ever be useful, but it is specified
by the ANSI C standard, so you may need to understand it.

Separate Expansion of Macro Arguments

```
        Separate Expansion of Macro Arguments
        ....................................
```

   We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

   What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

   The result is that the actual arguments are scanned *twice* to expand macro calls in them.

   Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

   You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro: the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self- references that do not expand in the first scan are marked so that they will not expand in the second scan either.

   The prescan is not done when an argument is stringified or concatenated. Thus,

```
    #define str(s) #s
    #define foo 4
    str (foo)
```

expands to `"foo"'. Once more, prescan has been prevented from having any noticeable effect.

   More precisely, stringification and concatenation use the argument as written, in un-prescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.

```
    #define str(s) #s lose(s)
    #define foo 4
    str (foo)
```

```
    expands to `"foo" lose(4)'.
```

   You might now ask, "Why mention the prescan, if it makes no difference?   And why not skip it and make the preprocessor faster?" The answer is that the prescan does make a difference in three special cases:

   * Nested calls to a macro.

   * Macros that call other macros that stringify or concatenate.

   * Macros whose expansions contain unshielded commas.

   We say that "nested" calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if `f' is a macro that expects one argument, `f (f (1))' is a nested pair of calls to `f'.  The desired expansion is made by expanding `f (1)' and substituting that into the definition of `f'.  The prescan causes the expected result to happen.  Without the prescan, `f (1)' itself would be substituted as an actual argument, and the inner use of `f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

   But prescan causes trouble in certain other cases of nested macro calls. Here is an example:

```
    #define foo  a,b
    #define bar(x) lose(x)
    #define lose(x) (1 + (x))

    bar(foo)
```

We would like `bar(foo)' to turn into `(1 + (foo))', which would then turn into `(1 + (a,b))'.  But instead, `bar(foo)' expands into `lose(a,b)', and you get an error because `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)
    #define bar(x) lose((x))
```

   The problem is more serious when the operands of the macro are not expressions; for example, when they are statements. Then parentheses are unacceptable because they would make for invalid C code:

```
    #define foo { int a, b; ... }
```

   There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it--if you use two levels of macros. Let's add a new macro `xstr' to the example shown above:

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

  This expands into \`"4"', not \`"foo"'. The reason for the difference is that
the argument of \`xstr' is expanded at prescan (because \`xstr' does not specify
stringification or concatenation of the argument). The result of prescan then
forms the actual argument for \`str'. \`str' uses its argument without prescan
because it performs stringification; but it cannot prevent or undo the
prescanning already done by \`xstr'.

The #elif Directive

```
                    The `#elif' Directive
                    .....................
```

   One common case of nested conditionals is used to check for more than two
possible alternatives. For example, you might have

```
     #if X == 1
     ...
     #else /* X != 1 */
     #if X == 2
     ...
     #else /* X != 2 */
     ...
     #endif /* X != 2 */
     #endif /* X != 1 */
```

   Another conditional directive, `#elif', allows this to be abbreviated as
follows:

```
     #if X == 1
     ...
     #elif X == 2
     ...
     #else /* X != 2 and X != 1*/
     ...
     #endif /* X != 2 and X != 1*/
```

   `#elif' stands for "else if".  Like `#else', it goes in the middle of a
`#if'-`#endif' pair and subdivides it; it does not require a matching `#endif'
of its own. Like `#if', the `#elif' directive includes an expression to be
tested.

   The text following the `#elif' is processed only if the original
`#if'-condition failed and the `#elif' condition succeeds. More than one
`#elif' can go in the same `#if'-`#endif' group. Then the text after each
`#elif' is processed only if the `#elif' condition succeeds after the original
`#if' and any previous `#elif' directives within it have failed. `#else' is
equivalent to `#elif 1', and `#else' is allowed after any number of `#elif'
directives, but `#elif' may not follow `#else'.

The #else Directive

```
                  The `#else' Directive
                  .....................
```

   The `#else' directive can be added to a conditional to provide alternative
text to be used if the condition is false.   This is what it looks like:

```
     #if EXPRESSION
     TEXT-IF-TRUE
     #else /* Not EXPRESSION */
     TEXT-IF-FALSE
     #endif /* Not EXPRESSION */
```

   If EXPRESSION is nonzero, and thus the TEXT-IF-TRUE is active, then `#else'
acts like a failing conditional and the TEXT-IF-FALSE is ignored.
Contrariwise, if the `#if' conditional fails, the TEXT-IF-FALSE is considered
included.

HTML Conversion by *AG2HTML.pl* V2.951201, perl 5.003 & *witbrock@cs.cmu.edu*

The #if' Directive

```
                        The `#if' Directive
                        ...................
```

   The `#if' directive in its simplest form consists of

```
     #if EXPRESSION
     CONTROLLED TEXT
     #endif /* EXPRESSION */
```

   The comment following the `#endif' is not required, but it is a good
practice because it helps people match the `#endif' to the corresponding
`#if'. Such comments should always be used, except in short conditionals that
are not nested. In fact, you can put anything at all after the `#endif' and
it will be ignored by the GNU C preprocessor, but only comments are acceptable
in ANSI Standard C.

   EXPRESSION is a C expression of integer type, subject to stringent
restrictions. It may contain

   * Integer constants, which are all regarded as `long' or `unsigned
     long'.

   * Character constants, which are interpreted according to the
     character set and conventions of the machine and operating system
     on which the preprocessor is running.

   * Arithmetic operators for addition, subtraction, multiplication,
     division, bitwise operations, shifts, comparisons, and logical
     operations (`&&' and `||').

   * Identifiers that are not macros, which are all treated as zero(!).

   * Macro calls. All macro calls in the expression are expanded before
     actual computation of the expression's value begins.

   Note that `sizeof' operators and `enum'-type values are not allowed
according to ANSI, but the Frexx C preprocessor allows the use of sizeof() and
even enables the user to set the sizeof size from the command line.
`enum'-type values, like all other identifiers that are not taken as macro
calls and expanded, are treated as zero.

   The CONTROLLED TEXT inside of a conditional can include preprocessing
directives. Then the directives inside the conditional are obeyed only if that
branch of the conditional succeeds. The text can also contain other
conditional groups. However, the `#if' and `#endif' directives must balance.

Unintended Grouping of Arithmetic

```
                    Unintended Grouping of Arithmetic
                    .................................
```

   You may have noticed that in most of the macro definition examples shown
above, each occurrence of a macro argument name had parentheses around it. In
addition, another pair of parentheses usually surround the entire macro
definition. Here is why it is best to write macros that way.

    Suppose you define a macro as follows,

```
    #define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to
compute how many `int' objects are needed to hold a certain number of `char'
objects.) Then suppose it is used as follows:

```
    a = ceil_div (b & c, sizeof (int));
```

This expands into

```
    a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make
it equivalent to this:

```
    a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
    a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
    #define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

   However, unintended grouping can result in another way. Consider `sizeof
ceil_div(1, 2)'. That has the appearance of a C expression that would compute
the size of the type of `ceil_div (1, 2)', but in fact it means something very
different. Here is what it expands to:

```
    sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence
rules have put the division outside the `sizeof' when it was intended to be

inside.

  Parentheses around the entire macro definition can prevent such problems.
Here, then, is the recommended way to define `ceil_div':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```