

## Chapter 14

# The Preprocessor

## Introduction

- Directives such as `#define` and `#include` are handled by the ***preprocessor***, a piece of software that edits C programs just prior to compilation.
- Its reliance on a preprocessor makes C (along with C++) unique among major programming languages. ( 優點 )
- The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs. ( 缺點 )

## How the Preprocessor Works

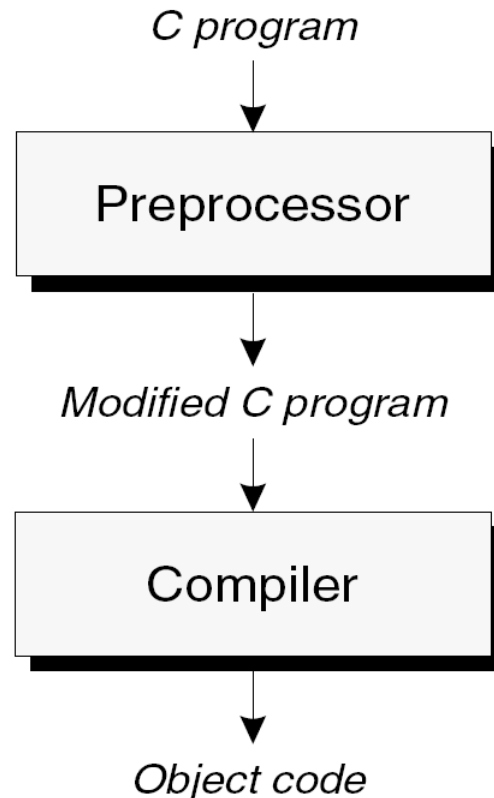
- The preprocessor looks for *preprocessing directives*, which begin with a # character.
- We've encountered the `#define` and `#include` directives before.
- `#define` defines a **macro**—a name that represents something else, such as a constant.
- The preprocessor responds to a `#define` directive by storing the name of the macro along with its definition.
- When the macro is used later, the preprocessor “expands” the macro, replacing it by its defined value.

## How the Preprocessor Works

- `#include` tells the preprocessor to open a particular file and “include” its contents as part of the file being compiled.
- For example, the line  
`#include <stdio.h>`  
instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program.

## How the Preprocessor Works

- The preprocessor's role in the compilation process:



## How the Preprocessor Works

- The input to the preprocessor is a C program, possibly containing directives.
- The preprocessor executes these directives, removing them in the process.
- The preprocessor's output goes directly into the compiler.

## How the Preprocessor Works

- The `celsius.c` program of Chapter 2:

```
/* Converts a Fahrenheit temperature to Celsius */  
  
#include <stdio.h>  
  
#define FREEZING_PT 32.0f  
#define SCALE_FACTOR (5.0f / 9.0f)  
  
int main(void)  
{  
    float fahrenheit, celsius;  
  
    printf("Enter Fahrenheit temperature: ");  
    scanf("%f", &fahrenheit);  
  
    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;  
    printf("Celsius equivalent is: %.1f\n", celsius);  
  
    return 0;  
}
```

## How the Preprocessor Works

- The program after preprocessing:

*Blank line*

*Blank line*

*Lines brought in from stdio.h*

*Blank line*

*Blank line*

*Blank line*

*Blank line*

```
int main(void)
{
    float fahrenheit, celsius;

    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);
    printf("Celsius equivalent is: %.1f\n", celsius);

    return 0;
}
```



## How the Preprocessor Works

- The preprocessor does a bit more than just execute directives.
- In particular, it replaces each comment with a single space character.
- Some preprocessors go further and remove unnecessary white-space characters, including spaces and tabs at the beginning of indented lines.

## How the Preprocessor Works

- In the early days of C, the preprocessor was a separate program.
- Nowadays, the preprocessor is often part of the compiler, and some of its output may not necessarily be C code.
- Still, it's useful to think of the preprocessor as separate from the compiler.

## How the Preprocessor Works

- Most C compilers provide a way to view the output of the preprocessor.
- Some compilers generate preprocessor output when a certain option is specified (GCC will do so when the `-E` option is used).
- Others come with a separate program that behaves like the integrated preprocessor.

## How the Preprocessor Works

- A word of caution: The preprocessor has only a limited knowledge of C.
- As a result, it's quite capable of creating illegal programs as it executes directives.
- In complicated programs, examining the output of the preprocessor may prove useful for locating this kind of error.

## Preprocessing Directives

- Most preprocessing directives fall into one of three categories:
  - **Macro definition.** The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
  - **File inclusion.** The `#include` directive causes the contents of a specified file to be included in a program.
  - **Conditional compilation.** The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program.

## Preprocessing Directives

- Several rules apply to all directives.
- ***Directives always begin with the # symbol.***  
The # symbol need not be at the beginning of a line, as long as only white space precedes it.
- ***Any number of spaces and horizontal tab characters may separate the tokens in a directive.***

Example:

```
#      define      N      100
```

## Preprocessing Directives

- *Directives always end at the first new-line character, unless explicitly continued.*

To continue a directive to the next line, end the current line with a \ character:

```
#define DISK_CAPACITY (SIDES *  
                        TRACKS_PER_SIDE *  
                        SECTORS_PER_TRACK *  
                        BYTES_PER_SECTOR)
```

## Preprocessing Directives

- ***Directives can appear anywhere in a program.***

Although `#define` and `#include` directives usually appear at the beginning of a file, other directives are more likely to show up later.

- ***Comments may appear on the same line as a directive.***

It's good practice to put a comment at the end of a macro definition:

```
#define FREEZING_PT 32.0f /* freezing point of water */
```



## Macro Definitions

- The macros that we've been using since Chapter 2 are known as *simple* macros, because they have no parameters.
- The preprocessor also supports *parameterized* macros.

## Simple Macros

- Definition of a *simple macro* (or *object-like macro*):

`#define identifier replacement-list`

*replacement-list* is any sequence of **preprocessing tokens**.

- The replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation.
- Wherever *identifier* appears later in the file, the preprocessor substitutes *replacement-list*.

## Simple Macros

- Any extra symbols in a macro definition will become part of the replacement list.
- Putting the = symbol in a macro definition is a common error:

```
#define N = 100    /* ** WRONG ** */
```

```
...
```

```
int a[N];          /* becomes int a[= 100]; */
```

## Simple Macros

- Ending a macro definition with a semicolon is another popular mistake:

```
#define N 100;    /** WRONG **/
```

```
...
```

```
int a[N];        /* becomes int a[100;];  
*/
```

- The compiler will detect most errors caused by extra symbols in a macro definition.
- Unfortunately, the compiler will flag each use of the macro as incorrect, rather than identifying the actual culprit: the macro's definition.

## Simple Macros

- Simple macros are primarily used for defining “manifest constants”—names that represent numeric, character, and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough
memory"
```

## Simple Macros

- Advantages of using `#define` to create names for constants:
  - ***It makes programs easier to read.*** The name of the macro can help the reader understand the meaning of the constant.
  - ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition.
  - ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like 3.14159 appears many times in a program, chances are it will occasionally be written 3.1416 or 3.14195 by accident.

## Simple Macros

- Simple macros have additional uses.
- ***Making minor changes to the syntax of C***

Macros can serve as alternate names for C symbols:

```
#define BEGIN {  
#define END   }  
#define LOOP for ( ;; )
```

Changing the syntax of C usually isn't a good idea, since it can make programs harder for others to understand.

## Simple Macros

- ***Renaming types***

An example from Chapter 5:

```
#define BOOL int
```

Type definitions are a better alternative.

- ***Controlling conditional compilation***

Macros play an important role in controlling conditional compilation.

A macro that might indicate “debugging mode”:

```
#define DEBUG
```



## Simple Macros

- When macros are used as constants, C programmers customarily capitalize all letters in their names.
- However, there's no consensus as to how to capitalize macros used for other purposes.
  - Some programmers like to draw attention to macros by using all upper-case letters in their names.
  - Others prefer lower-case names, following the style of K&R.

## Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):  
`#define identifier(  $x_1$  ,  $x_2$  , ... ,  $x_n$  ) replacement-list`  
 $x_1, x_2, \dots, x_n$  are identifiers (the macro's ***parameters***).
- The parameters may appear as many times as desired in the replacement list.
- There must be *no space* between the macro name and the left parenthesis.
- If space is left, the preprocessor will treat ( $x_1, x_2, \dots, x_n$ ) as part of the replacement list.

## Parameterized Macros

- When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use.
- Wherever a macro **invocation** of the form *identifier*( $y_1, y_2, \dots, y_n$ ) appears later in the program, the preprocessor replaces it with *replacement-list*, substituting  $y_1$  for  $x_1$ ,  $y_2$  for  $x_2$ , and so forth.
- Parameterized macros often serve as simple functions.

## Parameterized Macros

- Examples of parameterized macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))  
#define IS_EVEN(n) ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);  
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));  
if (((i)%2==0)) i++;
```

## Parameterized Macros

- A more complicated function-like macro:

```
#define TOUPPER(c) \
    ( 'a' <= (c) && (c) <= 'z' ? (c) - 'a' + 'A' : (c) )
```

- The `<ctype.h>` header provides a similar function named `toupper` that's more portable.
- A parameterized macro may have an empty parameter list:  

```
#define getchar() getc(stdin)
```
- The empty parameter list isn't really needed, but it makes `getchar` resemble a function.

## Parameterized Macros

- Using a parameterized macro instead of a true function has a couple of advantages:
  - ***The program may be slightly faster.*** A function call usually requires some overhead during program execution, but a macro invocation does not.
  - ***Macros are “generic.”*** A macro can **accept arguments of any type**, provided that the resulting program is valid.

## Parameterized Macros

- Parameterized macros also have disadvantages.
- ***The compiled code will often be larger.***

Each macro invocation increases the size of the source program (and hence the compiled code).

The problem is compounded when macro invocations are nested:

```
n = MAX(i, MAX(j, k));
```

The statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k)))));
```

## Parameterized Macros

- ***Arguments aren't type-checked.***

When a function is called, the compiler checks each argument to see if it has the appropriate type.

Macro arguments aren't checked by the preprocessor, nor are they converted.

- ***It's not possible to have a pointer to a macro.***

C allows **pointers to functions**, a useful concept.

Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro."



## Parameterized Macros

- *A macro may evaluate its arguments more than once.*

Unexpected behavior may occur if an argument has side effects:

```
n = MAX(i++, j);
```

The same line after preprocessing:

```
n = ((i++) > (j)) ? (i++) : (j);
```

If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

## Parameterized Macros

- Errors caused by evaluating a macro argument more than once can be difficult to find, because a macro invocation looks the same as a function call.
- To make matters worse, a macro may work properly most of the time, failing only for certain arguments that have side effects.
- For self-protection, it's a good idea to avoid side effects in arguments.

## Parameterized Macros

- Parameterized macros can be used as patterns for segments of code that are often repeated.
- A macro that makes it easier to display integers:  
`#define PRINT_INT(n) printf("%d\n", n)`
- The preprocessor will turn the line  
`PRINT_INT(i/j);`  
into  
`printf("%d\n", i/j);`

## The # Operator

- Macro definitions may contain two special operators, # and ##.
- Neither operator is recognized by the compiler; instead, they're executed during preprocessing.
- The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro.
- The operation performed by # is known as “stringization.”

## The # Operator

- There are a number of uses for #; let's consider just one.
- Suppose that we decide to use the `PRINT_INT` macro during debugging as a convenient way to print the values of integer variables and expressions.
- The `#` operator makes it possible for `PRINT_INT` to label each value that it prints.

## The # Operator

- Our new version of PRINT\_INT:  
`#define PRINT_INT(n) printf(#n " = %d\n", n)`
- The invocation  
`PRINT_INT(i/j);`  
will become  
`printf("i/j " " = %d\n", i/j);`
- The compiler automatically joins adjacent string literals, so this statement is equivalent to  
`printf("i/j = %d\n", i/j);`

## The ## Operator

- The ## operator can “paste” two tokens together to form a single token.
- If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument.

## The ## Operator

- A macro that uses the ## operator:  
`#define MK_ID(n) i##n`
- A declaration that invokes MK\_ID three times:  
`int MK_ID(1), MK_ID(2), MK_ID(3);`
- The declaration after preprocessing:  
`int i1, i2, i3;`



## The ## Operator

- The ## operator has a variety of uses.
- Consider the problem of defining a `max` function that behaves like the `MAX` macro described earlier.
- A single `max` function usually isn't enough, because it will only work for arguments of one type.
- Instead, we can write a macro that expands into the definition of a `max` function.
- The macro's parameter will specify the type of the arguments and the return value.

## The ## Operator

- There's just one snag: if we use the macro to create more than one function named `max`, the program won't compile.
- To solve this problem, we'll use the `##` operator to create a different name for each version of `max`:

```
#define GENERIC_MAX(type)      \  
type type##_max(type x, type y) \  
{                               \  
    return x > y ? x : y;      \  
}
```

- An invocation of this macro:

```
GENERIC_MAX(float)
```

- The resulting function definition:

```
float float_max(float x, float y) { return x > y ? x : y; }
```

## General Properties of Macros

- Several rules apply to both simple and parameterized macros.
- ***A macro's replacement list may contain invocations of other macros.***

Example:

```
#define PI      3.14159  
#define TWO_PI (2*PI)
```

When it encounters `TWO_PI` later in the program, the preprocessor replaces it by `(2*PI)`.

The preprocessor then ***rescans*** the replacement list to see if it contains invocations of other macros.

## General Properties of Macros

- ***The preprocessor replaces only entire tokens.***

Macro names embedded in identifiers, character constants, and string literals are ignored.

Example:

```
#define SIZE 256
```

```
int BUFFER_SIZE;
```

```
if (BUFFER_SIZE > SIZE)  
    puts("Error: SIZE exceeded");
```

Appearance after preprocessing:

```
int BUFFER_SIZE;
```

```
if (BUFFER_SIZE > 256)  
    puts("Error: SIZE exceeded");
```

## General Properties of Macros

- *A macro definition normally remains in effect until the end of the file in which it appears.*

Macros don't obey normal scope rules.

A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.

- *A macro may not be defined twice unless the new definition is identical to the old one.*

Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same.

## General Properties of Macros

- ***Macros may be “undefined” by the `#undef` directive.***

The `#undef` directive has the form

`#undef identifier`

where *identifier* is a macro name.

One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

## Parentheses in Macro Definitions

- The replacement lists in macro definitions often require parentheses in order to avoid unexpected results.
- If the macro's replacement list contains an operator, always enclose the replacement list in parentheses:  
`#define TWO_PI (2*3.14159)`
- Also, put parentheses around each parameter every time it appears in the replacement list:  
`#define SCALE(x) ((x)*10)`
- Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions.

## Parentheses in Macro Definitions

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159  
/* needs parentheses around replacement list */
```

- During preprocessing, the statement  
`conversion_factor = 360/TWO_PI;`

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication.



## Parentheses in Macro Definitions

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
    /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

This statement is equivalent to

```
j = i+10;
```

## Creating Longer Macros

- The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions.
- A macro that reads a string and then prints it:  
`#define ECHO(s) (gets(s), puts(s))`
- Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator.
- We can invoke `ECHO` as though it were a function:  
`ECHO(str); /* becomes (gets(str), puts(str)); */`

## Creating Longer Macros

- An alternative definition of `ECHO` that uses braces:

```
#define ECHO(s) { gets(s); puts(s); }
```

- Suppose that we use `ECHO` in an `if` statement:

```
if (echo_flag)
    ECHO(str);
else
    gets(str);
```

- Replacing `ECHO` gives the following result:

```
if (echo_flag)
    { gets(str); puts(str); };
else
    gets(str);
```

## Creating Longer Macros

- The compiler treats the first two lines as a complete `if` statement:

```
if (echo_flag)
    { gets(str); puts(str); }
```

- It treats the semicolon that follows as a null statement and produces an error message for the `else` clause, since it doesn't belong to any `if`.
- We could solve the problem by remembering not to put a semicolon after each invocation of `ECHO`, but then the program would look odd.

## Creating Longer Macros

- The comma operator solves this problem for `ECHO`, but not for all macros.
- If a macro needs to contain a series of *statements*, not just a series of *expressions*, the comma operator is of no help.
- The solution is to wrap the statements in a `do` loop whose condition is false:  

```
do { ... } while (0)
```
- Notice that the `do` statement needs a semicolon at the end.

## Creating Longer Macros

- A modified version of the ECHO macro:

```
#define ECHO(s)      \  
    do {            \  
        gets(s);    \  
        puts(s);    \  
    } while (0)
```

- When ECHO is used, it must be followed by a semicolon, which completes the do statement:

```
ECHO(str);  
/* becomes  
    do { gets(str); puts(str); } while (0); */
```

## Predefined Macros

- C has several predefined macros, each of which represents an integer constant or string literal.
- The `__DATE__` and `__TIME__` macros identify when a program was compiled.
- Example of using `__DATE__` and `__TIME__`:  

```
printf("Wacky Windows (c) 2010 Wacky Software,  
Inc.\n");  
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```
- Output produced by these statements:  
Wacky Windows (c) 2010 Wacky Software, Inc.  
Compiled on Dec 23 2010 at 22:18:48
- This information can be helpful for distinguishing among different versions of the same program.

## Predefined Macros

- We can use the `__LINE__` and `__FILE__` macros to help locate errors.
- A macro that can help pinpoint the location of a division by zero:

```
#define CHECK_ZERO(divisor) \  
    if (divisor == 0) \  
        printf("*** Attempt to divide by zero on line %d " \  
              "of file %s ***\n", __LINE__, __FILE__)
```

- The `CHECK_ZERO` macro would be invoked prior to a division:

```
CHECK_ZERO(j);  
k = i / j;
```



## Predefined Macros

- If `j` happens to be zero, a message of the following form will be printed:  

```
*** Attempt to divide by zero on line 9 of file foo.c ***
```
- Error-detecting macros like this one are quite useful.
- In fact, the C library has a general-purpose error-detecting macro named `assert`.
- The remaining predefined macro is named `__STDC__`.
- This macro exists and has the value 1 if the compiler conforms to the C standard (either C89 or C99).

## Additional Predefined Macros in C99

- C99 provides a few additional predefined macros.
- The `__STDC__HOSTED__` macro represents the constant 1 if the compiler is a hosted implementation. Otherwise, the macro has the value 0.
- An **implementation** of C consists of the compiler plus other software necessary to execute C programs.
- A **hosted implementation** must accept any program that conforms to the C99 standard.
- A **freestanding implementation** doesn't have to compile programs that use complex types or standard headers beyond a few of the most basic.

## Additional Predefined Macros in C99

- The `__STDC__VERSION__` macro provides a way to check which version of the C standard is recognized by the compiler.
  - If a compiler conforms to the C89 standard, including Amendment 1, the value is `199409L`.
  - If a compiler conforms to the C99 standard, the value is `199901L`.

## Additional Predefined Macros in C99

- A C99 compiler will define up to three additional macros, but only if the compiler meets certain requirements:

`__STDC_IEC_559__` is defined (and has the value 1) if the compiler performs floating-point arithmetic according to IEC 60559.

`__STDC_IEC_559_COMPLEX__` is defined (and has the value 1) if the compiler performs complex arithmetic according to IEC 60559.

`__STDC_ISO_10646__` is defined as `yyymmL` if wide characters are represented by the codes in ISO/IEC 10646 (with revisions as of the specified year and month).

## Empty Macro Arguments (C99)

- C99 allows any or all of the arguments in a macro call to be empty.
- Such a call will contain the same number of commas as a normal call.
- Wherever the corresponding parameter name appears in the replacement list, it's replaced by nothing.

## Empty Macro Arguments (C99)

- Example:  
`#define ADD(x,y) (x+y)`
- After preprocessing, the statement  
`i = ADD(j,k);`  
becomes  
`i = (j+k);`  
whereas the statement  
`i = ADD(,k);`  
becomes  
`i = (+k);`

## Empty Macro Arguments (C99)

- When an empty argument is an operand of the # or ## operators, special rules apply.
- If an empty argument is “stringized” by the # operator, the result is "" (the empty string):

```
#define MK_STR(x) #x
```

...

```
char empty_string[] = MK_STR();
```

- The declaration after preprocessing:

```
char empty_string[] = "";
```

## Empty Macro Arguments (C99)

- If one of the arguments of the `##` operator is empty, it's replaced by an invisible “placemaker” token.
- Concatenating an ordinary token with a placemaker token yields the original token (the placemaker disappears).
- If two placemaker tokens are concatenated, the result is a single placemaker.
- Once macro expansion has been completed, placemaker tokens disappear from the program.



## Empty Macro Arguments (C99)

- Example:

```
#define JOIN(x,y,z) x##y##z
```

...

```
int JOIN(a,b,c), JOIN(a,b, ), JOIN(a, , c), JOIN( , , c);
```

- The declaration after preprocessing:  

```
int abc, ab, ac, c;
```
- The missing arguments were replaced by placemaker tokens, which then disappeared when concatenated with any nonempty arguments.
- All three arguments to the `JOIN` macro could even be missing, which would yield an empty result.

## Macros with a Variable Number of Arguments (C99)

- C99 allows macros that take an unlimited number of arguments.
- A macro of this kind can pass its arguments to a function that accepts a variable number of arguments.

- Example:

```
#define TEST(condition, ...) ((condition)? \
    printf("Passed test: %s\n", #condition): \
    printf(__VA_ARGS__))
```

- The . . . token (*ellipsis*) goes at the end of the parameter list, preceded by ordinary parameters, if any.
- `__VA_ARGS__` is a special identifier that represents all the arguments that correspond to the ellipsis.

## Macros with a Variable Number of Arguments (C99)

- An example that uses the TEST macro:

```
TEST(voltage <= max_voltage,  
    "Voltage %d exceeds %d\n", voltage, max_voltage);
```

- Preprocessor output (reformatted for readability):

```
((voltage <= max_voltage)?  
    printf("Passed test: %s\n", "voltage <= max_voltage"):  
    printf("Voltage %d exceeds %d\n", voltage, max_voltage));
```

- The program will display the message  
Passed test: voltage <= max\_voltage  
if voltage is no more than max\_voltage.
- Otherwise, it will display the values of voltage and max\_voltage:  
Voltage 125 exceeds 120

## The `__func__` Identifier (C99)

- The `__func__` identifier behaves like a string variable that stores the name of the currently executing function.
- The effect is the same as if each function contains the following declaration at the beginning of its body:  
`static const char __func__[] = "function-name";`  
where *function-name* is the name of the function.

## The `__func__` Identifier (C99)

- Debugging macros that rely on the `__func__` identifier:  

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);  
#define FUNCTION_RETURNS() printf("%s returns\n", __func__);
```
- These macros can be used to trace function calls:  

```
void f(void)  
{  
    FUNCTION_CALLED();    /* displays "f called" */  
    ...  
    FUNCTION_RETURNS();  /* displays "f returns" */  
}
```
- Another use of `__func__`: it can be passed to a function to let it know the name of the function that called it.

## Conditional Compilation

- The C preprocessor recognizes a number of directives that support ***conditional compilation***.
- This feature permits the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

## The `#if` and `#endif` Directives

- Suppose we're in the process of debugging a program.
- We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program.
- Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later.
- Conditional compilation allows us to leave the calls in place, but have the compiler ignore them.

## The `#if` and `#endif` Directives

- The first step is to define a macro and give it a nonzero value:  
`#define DEBUG 1`
- Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```



## The `#if` and `#endif` Directives

- During preprocessing, the `#if` directive will test the value of `DEBUG`.
- Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program.
- If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program.
- The `#if-#endif` blocks can be left in the final program, allowing diagnostic information to be produced later if any problems turn up.

## The `#if` and `#endif` Directives

- General form of the `#if` and `#endif` directives:  
`#if` *constant-expression*  
`#endif`
- When the preprocessor encounters the `#if` directive, it evaluates the constant expression.
- If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing.
- Otherwise, the lines between `#if` and `#endif` will remain.

## The **#if** and **#endif** Directives

- The **#if** directive treats undefined identifiers as macros that have the value 0.
- If we neglect to define **DEBUG**, the test  
`#if DEBUG`  
will fail (but not generate an error message).
- The test  
`#if !DEBUG`  
will succeed.

## The `defined` Operator

- The preprocessor supports three operators: `#`, `##`, and `defined`.
- When applied to an identifier, `defined` produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise.
- The `defined` operator is normally used in conjunction with the `#if` directive.

## The `defined` Operator

- Example:  
`#if defined(DEBUG)`  
...  
`#endif`
- The lines between `#if` and `#endif` will be included only if `DEBUG` is defined as a macro.
- The parentheses around `DEBUG` aren't required:  
`#if defined DEBUG`
- It's not necessary to give `DEBUG` a value:  
`#define DEBUG`

## The `#ifdef` and `#ifndef` Directives

- The `#ifdef` directive tests whether an identifier is currently defined as a macro:  
`#ifdef identifier`
- The effect is the same as  
`#if defined(identifier)`
- The `#ifndef` directive tests whether an identifier is *not* currently defined as a macro:  
`#ifndef identifier`
- The effect is the same as  
`#if !defined(identifier)`

## The **#elif** and **#else** Directives

- **#if**, **#ifdef**, and **#ifndef** blocks can be nested just like ordinary **if** statements.
- When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows.
- Some programmers put a comment on each closing **#endif** to indicate what condition the matching **#if** tests:

```
#if DEBUG
...
#endif /* DEBUG */
```

## The `#elif` and `#else` Directives

- `#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

`#if expr1`

*Lines to be included if `expr1` is nonzero*

`#elif expr2`

*Lines to be included if `expr1` is zero but `expr2` is nonzero*

`#else`

*Lines to be included otherwise*

`#endif`

- Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.



## Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging.
- ***Writing programs that are portable to several machines or operating systems.***

Example:

```
#if defined(WIN32)
...
#elif defined(MAC_OS)
...
#elif defined(LINUX)
...
#endif
```

## Uses of Conditional Compilation

- *Writing programs that can be compiled with different compilers.*

An example that uses the `__STDC__` macro:

```
#if __STDC__
```

```
    Function prototypes
```

```
#else
```

```
    Old-style function declarations
```

```
#endif
```

If the compiler does not conform to the C standard, old-style function declarations are used instead of function prototypes.

## Uses of Conditional Compilation

- ***Providing a default definition for a macro.***

Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

## Uses of Conditional Compilation

- *Temporarily disabling code that contains comments.*

A `/* ... */` comment can't be used to “comment out” code that already contains `/* ... */` comments.

An `#if` directive can be used instead:

```
#if 0
```

*Lines containing comments*

```
#endif
```

## Uses of Conditional Compilation

- Chapter 15 discusses another common use of conditional compilation: protecting header files against multiple inclusion.

## Miscellaneous Directives

- The `#error`, `#line`, and `#pragma` directives are more specialized than the ones we've already examined.
- These directives are used much less frequently.

## The `#error` Directive

- Form of the `#error` directive:  
`#error message`  
*message* is any sequence of tokens.
- If the preprocessor encounters an `#error` directive, it prints an error message which must include *message*.
- If an `#error` directive is processed, some compilers immediately terminate compilation without attempting to find other errors.

## The `#error` Directive

- `#error` directives are frequently used in conjunction with conditional compilation.
- Example that uses an `#error` directive to test the maximum value of the `int` type:

```
#if INT_MAX < 1000000
#error int type is too small
#endif
```



## The `#error` Directive

- The `#error` directive is often found in the `#else` part of an `#if-#elif-#else` series:

```
#if defined(WIN32)
```

```
...
```

```
#elif defined(MAC_OS)
```

```
...
```

```
#elif defined(LINUX)
```

```
...
```

```
#else
```

```
#error No operating system specified
```

```
#endif
```

## The `#line` Directive

- The `#line` directive is used to alter the way program lines are numbered.
- First form of the `#line` directive:

`#line n`

Subsequent lines in the program will be numbered  $n$ ,  $n + 1$ ,  $n + 2$ , and so forth.

- Second form of the `#line` directive:

`#line n "file"`

Subsequent lines are assumed to come from *file*, with line numbers starting at  $n$ .

## The `#line` Directive

- The `#line` directive changes the value of the `__LINE__` macro (and possibly `__FILE__`).
- Most compilers will use the information from the `#line` directive when generating error messages.
- Suppose that the following directive appears at the beginning of `foo.c`:

```
#line 10 "bar.c"
```

If the compiler detects an error on line 5 of `foo.c`, the message will refer to line 13 of file `bar.c`.

- The `#line` directive is used primarily by programs that generate C code as output.

## The `#line` Directive

- The most famous example is `yacc` (Yet Another Compiler-Compiler), a UNIX utility that automatically generates part of a compiler.
- The programmer prepares a file that contains information for `yacc` as well as fragments of C code.
- From this file, `yacc` generates a C program, `y.tab.c`, that incorporates the code supplied by the programmer.
- By inserting `#line` directives, `yacc` tricks the compiler into believing that the code comes from the original file.
- Error messages produced during the compilation of `y.tab.c` will refer to lines in the original file.

## The **#pragma** Directive

- The **#pragma** directive provides a way to request special behavior from the compiler.
- Form of a **#pragma** directive:  
`#pragma tokens`
- **#pragma** directives can be very simple (a single token) or they can be much more elaborate:  
`#pragma data(heap_size => 1000, stack_size => 2000)`

## The **#pragma** Directive

- The set of commands that can appear in **#pragma** directives is different for each compiler.
- The preprocessor must ignore any **#pragma** directive that contains an unrecognized command; it's not permitted to give an error message.
- In C89, there are no standard pragmas—they're all implementation-defined.
- C99 has three standard pragmas, all of which use **STDC** as the first token following **#pragma**.

## The `_Pragma` Operator (C99)

- C99 introduces the `_Pragma` operator, which is used in conjunction with the `#pragma` directive.
- A `_Pragma` expression has the form  
`_Pragma ( string-literal )`
- When it encounters such an expression, the preprocessor “destringizes” the string literal:
  - Double quotes around the string are removed.
  - `\"` is replaced by `"`.
  - `\\` is replaced by `\`.

## The `_Pragma` Operator (C99)

- The resulting tokens are then treated as though they appear in a `#pragma` directive.

- For example, writing

```
_Pragma("data(heap_size=>1000, stack_size=>2000)")
```

is the same as writing

```
#pragma data(heap_size=>1000, stack_size=>2000)
```



## The `_Pragma` Operator (C99)

- The `_Pragma` operator lets us work around the fact that a preprocessing directive can't generate another directive.
- `_Pragma`, however, is an operator, not a directive, and can therefore appear in a macro definition.
- This makes it possible for a macro expansion to leave behind a `#pragma` directive.

## The `_Pragma` Operator (C99)

- A macro that uses the `_Pragma` operator:  
`#define DO_PRAGMA(x) _Pragma(#x)`
- An invocation of the macro:  
`DO_PRAGMA(GCC dependency "parse.y")`
- The result after expansion:  
`#pragma GCC dependency "parse.y"`
- The tokens passed to `DO_PRAGMA` are stringized into `"GCC dependency \"parse.y\""`.
- The `_Pragma` operator destringizes this string, producing a `#pragma` directive.