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
- The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs

How the Preprocessor Works

- The preprocessor looks for ***preprocessing directives*,** which begin with a `#` character
- So far, we have encountered the `#define` and `#include` directives
- `#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

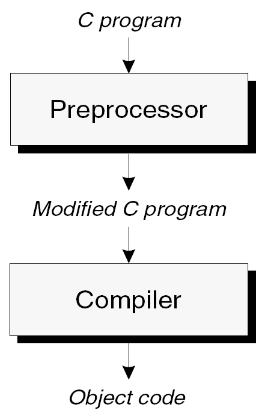
- `#include` tells the preprocessor to open a particular file and "*include*" its contents as part of the file being compiled
- For example, the line

  ```
  #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

```
/* 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
- In the early days of `C`, the preprocessor was a separate program

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` and `cc` will do so when the `-E` option is used)

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 (i.e., \ character immediately followed by a ***new-line***):

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


Chapter 14: The Preprocessor

Preprocessing Directives

- ***Directives can appear any where 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 is *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 have 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***
- It is *legal* to have *empty replacement-list*
- The replacement list may include:
  identifiers, keywords, numeric constants, punctuation, operators, character constants, and string literals
- Wherever the *identifier* appears ***later*** in the file, the preprocessor substitutes it with the *replacement-list*

Simple Macros

- When macros are used as constants, C programmers usually capitalize all letters in their names
- However, there is 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

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]; */
```

- 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

Simple Macros

- Simple macros are primarily used for defining "*manifest constants*", i.e., 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 *is not 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
```

Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):

  `#define` *identifier* `(`x_1 `,` x_2 `,` … `,` x_n`)` *replacement-list*

  $x_1, x_2, \ldots, x_n$ 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 a space is left between the macro name and the left parenthesis, the preprocessor will treat*** $(x_1, x_2, \ldots, x_n)$ ***as part of the replacement list***

Chapter 14: The Preprocessor

Parameterized Macros

- Wherever a macro ***invocation*** of the form *identifier* $(y_1, y_2, \ldots, y_n)$ 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) \
  ((c)>='a'&&(c)<='z'?(c)-'a'+'A':(c))
```

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

```
#define getchar() getc(stdin)
```

- The empty parameter list is not 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 are not type-checked***

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

Macro arguments are not checked by the preprocessor, nor are they converted

– ***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

Chapter 14: The Preprocessor

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 is *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);
```