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.
- 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.
- 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

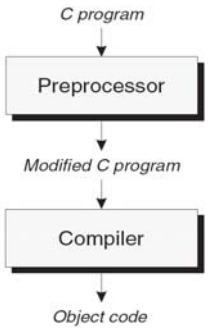
- `#include` tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled.
- For example, the line

  ```
  #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;
}
```


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

- ***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

- 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.

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++;
```


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 first step is to define a macro and give it a nonzero value:

```
#define DEBUG 1
```

- Next, surround a 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 `#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`.

Chapter 15

Writing Large Programs

Source Files

- A C program may be divided among any number of ***source files.***
- By convention, source files have the extension `.c`.
- Each source file contains part of the program, primarily definitions of functions and variables.
- One source file must contain a function named `main`, which serves as the starting point for the program.

Building a Multiple-File Program

- Building a large program requires the same basic steps as building a small one:
  - Compiling
  - Linking

Building a Multiple-File Program

- Each source file in the program must be compiled separately.
- Header files don't need to be compiled.
- The contents of a header file are automatically compiled whenever a source file that includes it is compiled.
- For each source file, the compiler generates a file containing object code.
- These files—known as ***object files***—have the extension `.o` in UNIX and `.obj` in Windows.

Building a Multiple-File Program

- The linker combines the object files created in the previous step—along with code for library functions—to produce an executable file.
- Among other duties, the linker is responsible for resolving external references left behind by the compiler.
- An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.

Building a Multiple-File Program

- Most compilers allow us to build a program in a single step.
- A GCC command that builds `justify`:

```
gcc -o justify justify.c line.c word.c
```

- The three source files are first compiled into object code.
- The object files are then automatically passed to the linker, which combines them into a single file.
- The `-o` option specifies that we want the executable file to be named `justify`.

Makefiles

- To make it easier to build large programs, UNIX originated the concept of the ***makefile.***
- A makefile not only lists the files that are part of the program, but also describes ***dependencies*** among the files.
- Suppose that the file `foo.c` includes the file `bar.h`.
- We say that `foo.c` "depends" on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

Makefiles

- A UNIX makefile for the `justify` program:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
        gcc -c justify.c

word.o: word.c word.h
        gcc -c word.c

line.o: line.c line.h
        gcc -c line.c
```


Makefiles

- There are four groups of lines; each group is known as a ***rule.***
- The first line in each rule gives a ***target*** file, followed by the files on which it depends.
- The second line is a ***command*** to be executed if the target should need to be rebuilt because of a change to one of its dependent files.

Makefiles

- In the first rule, `justify` (the executable file) is the target:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o
```

- The first line states that `justify` depends on the files `justify.o`, `word.o`, and `line.o`.
- If any of these files have changed since the program was last built, `justify` needs to be rebuilt.
- The command on the following line shows how the rebuilding is to be done.

Makefiles

- In the second rule, `justify.o` is the target:

```
justify.o: justify.c word.h line.h
        gcc -c justify.c
```

- The first line indicates that `justify.o` needs to be rebuilt if there's been a change to `justify.c`, `word.h`, or `line.h`.
- The next line shows how to update `justify.o` (by recompiling `justify.c`).
- The `-c` option tells the compiler to compile `justify.c` but not attempt to link it.

Makefiles

- Once we've created a makefile for a program, we can use the `make` utility to build (or rebuild) the program.
- By checking the time and date associated with each file in the program, `make` can determine which files are out of date.
- It then invokes the commands necessary to rebuild the program.

Makefiles

- Each command in a makefile must be preceded by a tab character, not a series of spaces.
- A makefile is normally stored in a file named `Makefile` (or `makefile`).
- When the `make` utility is used, it automatically checks the current directory for a file with one of these names.

Makefiles

- To invoke `make`, use the command

  `make` *target*

  where *target* is one of the targets listed in the makefile.
- If no target is specified when `make` is invoked, it will build the target of the first rule.
- Except for this special property of the first rule, the order of rules in a makefile is arbitrary.