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.

Source Files

- Consider the problem of writing a simple calculator program.
- The program will evaluate integer expressions entered in Reverse Polish notation (RPN), in which operators follow operands.
- If the user enters an expression such as

  ```
  30 5 - 7 *
  ```

  the program should print its value (175, in this case).

Source Files

- The program will read operands and operators, one by one, using a stack to keep track of intermediate results.
  - If the program reads a number, it will push the number onto the stack.
  - If the program reads an operator, it will pop two numbers from the stack, perform the operation, and then push the result back onto the stack.
- When the program reaches the end of the user's input, the value of the expression will be on the stack.

Source Files

- How the expression `30 5 - 7 *`: will be evaluated:
  1. Push 30 onto the stack.
  2. Push 5 onto the stack.
  3. Pop the top two numbers from the stack, subtract 5 from 30, giving 25, and then push the result back onto the stack.
  4. Push 7 onto the stack.
  5. Pop the top two numbers from the stack, multiply them, and then push the result back onto the stack.
- The stack will now contain 175, the value of the expression.

Source Files

- The program's `main` function will contain a loop that performs the following actions:
  - Read a "token" (a number or an operator).
  - If the token is a number, push it onto the stack.
  - If the token is an operator, pop its operands from the stack, perform the operation, and then push the result back onto the stack.
- When dividing a program like this one into files, it makes sense to put related functions and variables into the same file.

Source Files

- The function that reads tokens could go into one source file (`token.c`, say), together with any functions that have to do with tokens.
- Stack-related functions such as `push`, `pop`, `make_empty`, `is_empty`, and `is_full` could go into a different file, `stack.c`.
- The variables that represent the stack would also go into `stack.c`.
- The `main` function would go into yet another file, `calc.c`.

Source Files

- Splitting a program into multiple source files has significant advantages:
  - Grouping related functions and variables into a single file helps clarify the structure of the program.
  - Each source file can be compiled separately, which saves time.
  - Functions are more easily reused in other programs when grouped in separate source files.

Header Files

- Problems that arise when a program is divided into several source files:
  - How can a function in one file call a function that's defined in another file?
  - How can a function access an external variable in another file?
  - How can two files share the same macro definition or type definition?
- The answer lies with the `#include` directive, which makes it possible to share information among any number of source files.

Header Files

- The `#include` directive tells the preprocessor to insert the contents of a specified file.
- Information to be shared among several source files can be put into such a file.
- `#include` can then be used to bring the file's contents into each of the source files.
- Files that are included in this fashion are called ***header files*** (or sometimes ***include files***).
- By convention, header files have the extension `.h`.

The `#include` Directive

- The `#include` directive has two primary forms.
- The first is used for header files that belong to C's own library:

  `#include <`*filename*`>`
- The second is used for all other header files:

  `#include "`*filename*`"`
- The difference between the two has to do with how the compiler locates the header file.

The `#include` Directive

- Typical rules for locating header files:
  - `#include <`*filename*`>`: Search the directory (or directories) in which system header files reside.
  - `#include "`*filename*`"`: Search the current directory, then search the directory (or directories) in which system header files reside.
- The places to be searched for header files can usually be altered, often by a command-line option such as **`-I`*path***.

The **#include** Directive

- Don’t use brackets when including header files that you have written:

```
#include <myheader.h>   /*** WRONG ***/
```

- The preprocessor will probably look for `myheader.h` where the system header files are kept.

The `#include` Directive

- The file name in an `#include` directive may include information that helps locate the file, such as a directory path or drive specifier:

```
#include "c:\cprogs\utils.h"
  /* Windows path */

#include "/cprogs/utils.h"
  /* UNIX path */
```

- Although the quotation marks in the `#include` directive make file names look like string literals, the preprocessor doesn't treat them that way.

The `#include` Directive

- It's usually best not to include path or drive information in `#include` directives.
- Bad examples of Windows `#include` directives:

```
#include "d:utils.h"
#include "\cprogs\include\utils.h"
#include "d:\cprogs\include\utils.h"
```

- Better versions:

```
#include "utils.h"
#include "..\include\utils.h"
```

The `#include` Directive

- The `#include` directive has a third form:

  `#include` *tokens*

  tokens is any sequence of preprocessing tokens.
- The preprocessor will scan the tokens and replace any macros that it finds.
- After macro replacement, the resulting directive must match one of the other forms of `#include`.
- The advantage of the third kind of `#include` is that the file name can be defined by a macro rather than being "hard-coded" into the directive itself.

The **#include** Directive

- Example:

```
#if defined(IA32)
  #define CPU_FILE "ia32.h"
#elif defined(IA64)
  #define CPU_FILE "ia64.h"
#elif defined(AMD64)
  #define CPU_FILE "amd64.h"
#endif

#include CPU_FILE
```

Sharing Macro Definitions and Type Definitions

- Most large programs contain macro definitions and type definitions that need to be shared by several source files.
- These definitions should go into header files.

Sharing Macro Definitions and Type Definitions

- Suppose that a program uses macros named `BOOL`, `TRUE`, and `FALSE`.
- Their definitions can be put in a header file with a name like `boolean.h`:

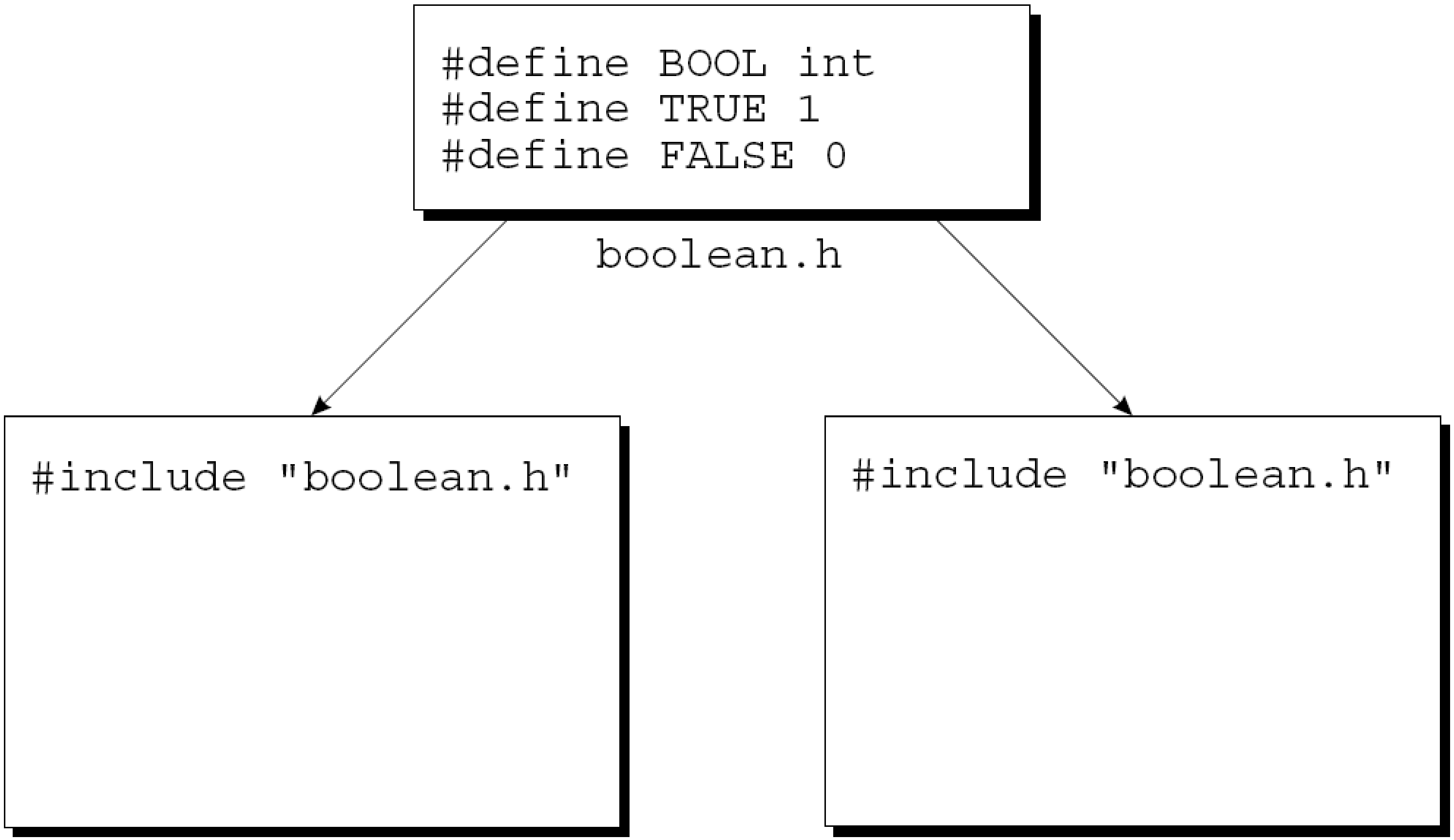
```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

- Any source file that requires these macros will simply contain the line

```
#include "boolean.h"
```

Sharing Macro Definitions and Type Definitions

- A program in which two files include `boolean.h`:



Sharing Macro Definitions and Type Definitions

- Type definitions are also common in header files.
- For example, instead of defining a `BOOL` macro, we might use `typedef` to create a `Bool` type.
- If we do, the `boolean.h` file will have the following appearance:

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

Sharing Macro Definitions and Type Definitions

- Advantages of putting definitions of macros and types in header files:
  - Saves time. We don't have to copy the definitions into the source files where they're needed.
  - Makes the program easier to modify. Changing the definition of a macro or type requires editing a single header file.
  - Avoids inconsistencies caused by source files containing different definitions of the same macro or type.

Sharing Function Prototypes

- Suppose that a source file contains a call of a function `f` that's defined in another file, `foo.c`.
- Calling `f` without declaring it first is risky.
  - The compiler assumes that `f`'s return type is `int`.
  - It also assumes that the number of parameters matches the number of arguments in the call of `f`.
  - The arguments themselves are converted automatically by the default argument promotions.

Sharing Function Prototypes

- Declaring `f` in the file where it's called solves the problem but can create a maintenance nightmare.
- A better solution is to put `f`'s prototype in a header file (`foo.h`), then include the header file in all the places where `f` is called.
- We'll also need to include `foo.h` in `foo.c`, enabling the compiler to check that `f`'s prototype in `foo.h` matches its definition in `foo.c`.

Sharing Function Prototypes

- If `foo.c` contains other functions, most of them should be declared in `foo.h`.
- Functions that are intended for use only within `foo.c` shouldn't be declared in a header file, however; to do so would be misleading.

Sharing Function Prototypes

- The RPN calculator example can be used to illustrate the use of function prototypes in header files.
- The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions.
- Prototypes for these functions should go in the `stack.h` header file:

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

Sharing Function Prototypes

- We'll include `stack.h` in `calc.c` to allow the compiler to check any calls of stack functions that appear in the latter file.
- We'll also include `stack.h` in `stack.c` so the compiler can verify that the prototypes in `stack.h` match the definitions in `stack.c`.

Sharing Function Prototypes

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

stack.h

```
#include "stack.h"

int main(void)
{
  make_empty();
  …
}
```

calc.c

```
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(void)
{ … }

int is_empty(void)
{ … }

int is_full(void)
{ … }

void push(int i)
{ … }

int pop(void)
{ … }
```

stack.c

Sharing Variable Declarations

- To share a function among files, we put its *definition* in one source file, then put *declarations* in other files that need to call the function.
- Sharing an external variable is done in much the same way.

Sharing Variable Declarations

- An example that both declares and defines `i` (causing the compiler to set aside space):

```
int i;
```

- The keyword `extern` is used to declare a variable without defining it:

```
extern int i;
```

- `extern` informs the compiler that `i` is defined elsewhere in the program, so there's no need to allocate space for it.

Sharing Variable Declarations

- When we use `extern` in the declaration of an array, we can omit the length of the array:

  ```
  extern int a[];
  ```

- Since the compiler doesn't allocate space for `a` at this time, there's no need for it to know `a`'s length.

Sharing Variable Declarations

- To share a variable `i` among several source files, we first put a definition of `i` in one file:

  ```
  int i;
  ```

- If `i` needs to be initialized, the initializer would go here.
- The other files will contain declarations of `i`:

  ```
  extern int i;
  ```

- By declaring `i` in each file, it becomes possible to access and/or modify `i` within those files.

Sharing Variable Declarations

- When declarations of the same variable appear in different files, the compiler can't check that the declarations match the variable's definition.
- For example, one file may contain the definition

  ```
  int i;
  ```

  while another file contains the declaration

  ```
  extern long i;
  ```

- An error of this kind can cause the program to behave unpredictably.

Sharing Variable Declarations

- To avoid inconsistency, declarations of shared variables are usually put in header files.
- A source file that needs access to a particular variable can then include the appropriate header file.
- In addition, each header file that contains a variable declaration is included in the source file that contains the variable's definition, enabling the compiler to check that the two match.

Nested Includes

- A header file may contain `#include` directives.
- `stack.h` contains the following prototypes:

```
int is_empty(void);
int is_full(void);
```

- Since these functions return only 0 or 1, it's a good idea to declare their return type to be `Bool`:

```
Bool is_empty(void);
Bool is_full(void);
```

- We'll need to include the `boolean.h` file in `stack.h` so that the definition of `Bool` is available when `stack.h` is compiled.

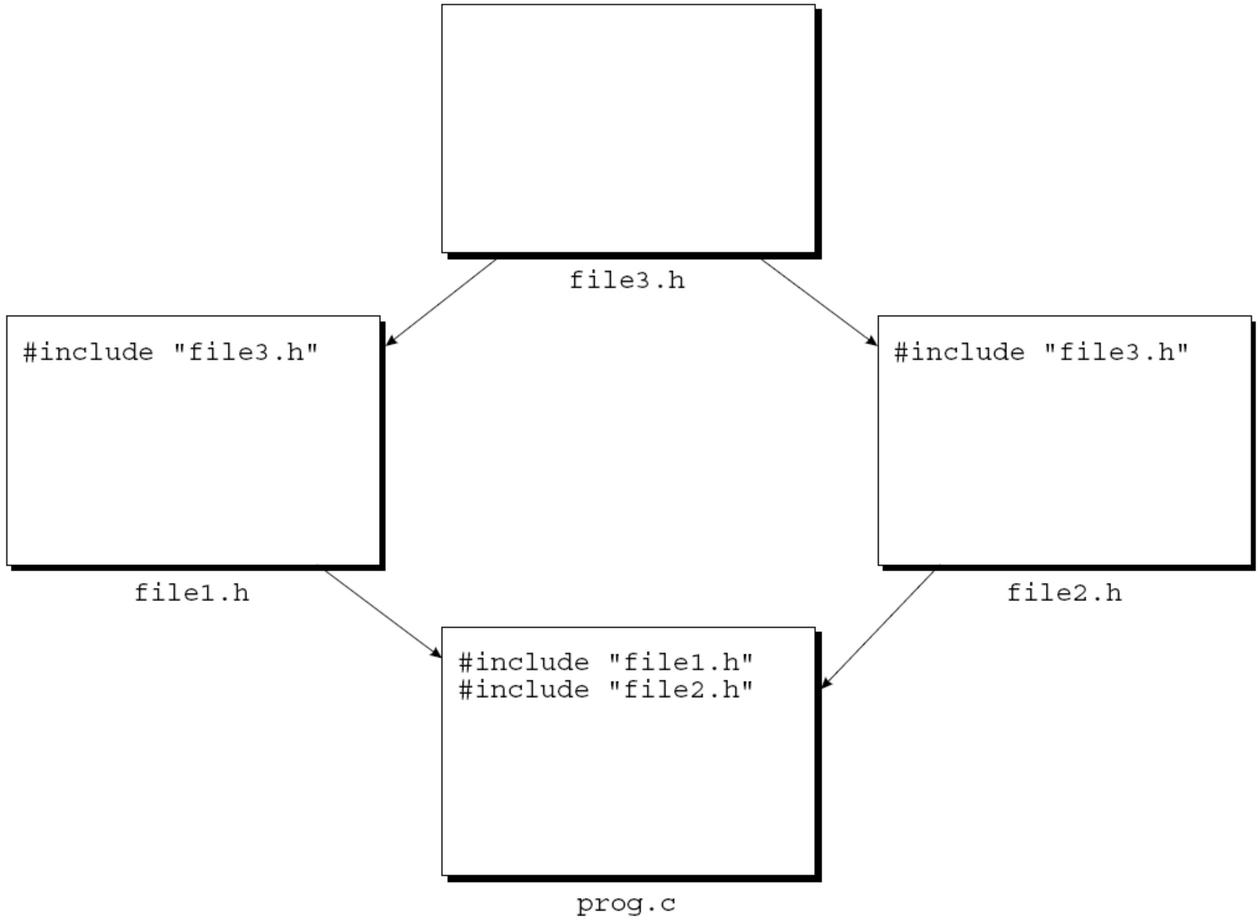
Nested Includes

- Traditionally, C programmers shun nested includes.
- However, the bias against nested includes has largely faded away, in part because nested includes are common practice in C++.

Protecting Header Files

- If a source file includes the same header file twice, compilation errors may result.
- This problem is common when header files include other header files.
- Suppose that `file1.h` includes `file3.h`, `file2.h` includes `file3.h`, and `prog.c` includes both `file1.h` and `file2.h`.

Protecting Header Files



- When `prog.c` is compiled, `file3.h` will be compiled twice.

Protecting Header Files

- Including the same header file twice doesn't always cause a compilation error.
- If the file contains only macro definitions, function prototypes, and/or variable declarations, there won't be any difficulty.
- If the file contains a type definition, however, we'll get a compilation error.

Protecting Header Files

- Just to be safe, it's probably a good idea to protect all header files against multiple inclusion.
- That way, we can add type definitions to a file later without the risk that we might forget to protect the file.
- In addition, we might save some time during program development by avoiding unnecessary recompilation of the same header file.

Protecting Header Files

- To protect a header file, we'll enclose the contents of the file in an `#ifndef`-`#endif` pair.
- How to protect the `boolean.h` file:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

Protecting Header Files

- Making name of the macro resemble the name of the header file is a good way to avoid conflicts with other macros.
- Since we can't name the macro `BOOLEAN.H`, a name such as `BOOLEAN_H` is a good alternative.

`#error` Directives in Header Files

- `#error` directives are often put in header files to check for conditions under which the header file shouldn't be included.
- Suppose that a header file uses a feature that didn't exist prior to the original C89 standard.
- An `#ifndef` directive that tests for the existence of the `__STDC__` macro:

```
#ifndef __STDC__
#error This header requires a Standard C compiler
#endif
```

Dividing a Program into Files

- Designing a program involves determining what functions it will need and arranging the functions into logically related groups.
- Once a program has been designed, there is a simple technique for dividing it into files.

Dividing a Program into Files

- Each set of functions will go into a separate source file (`foo.c`).
- Each source file will have a matching header file (`foo.h`).
  - `foo.h` will contain prototypes for the functions defined in `foo.c`.
  - Functions to be used only within `foo.c` should not be declared in `foo.h`.
- `foo.h` will be included in each source file that needs to call a function defined in `foo.c`.
- `foo.h` will also be included in `foo.c` so the compiler can check that the prototypes in `foo.h` match the definitions in `foo.c`.

Dividing a Program into Files

- The `main` function will go in a file whose name matches the name of the program.
- It's possible that there are other functions in the same file as `main`, so long as they're not called from other files in the program.

Program: Text Formatting

- Let's apply this technique to a small text-formatting program named `justify`.
- Assume that a file `quote` contains the following:

```
   C      is quirky,  flawed,    and  an
enormous   success.      Although accidents of   history
 surely  helped,   it evidently    satisfied   a   need

    for  a   system  implementation    language
  efficient
 enough   to  displace         assembly   language,
   yet sufficiently   abstract   and fluent    to
  describe
  algorithms   and     interactions    in a   wide
  variety
of   environments.
                          --      Dennis     M.        Ritchie
```

Program: Text Formatting

- To run the program from a UNIX or Windows prompt, we'd enter the command

```
justify <quote
```

- The `<` symbol informs the operating system that justify will read from the file `quote` instead of accepting input from the keyboard.
- This feature, supported by UNIX, Windows, and other operating systems, is called ***input redirection.***

Program: Text Formatting

- Output of `justify`:

```
C is quirky,  flawed,  and  an  enormous  success.  Although
accidents of history surely helped, it evidently satisfied a
need for a system implementation language  efficient  enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and  interactions  in  a  wide
variety of environments. -- Dennis M. Ritchie
```

- The output of `justify` will normally appear on the screen, but we can save it in a file by using ***output redirection:***

```
justify <quote >newquote
```

Program: Text Formatting

- `justify` will delete extra spaces and blank lines as well as filling and justifying lines.
  - "Filling" a line means adding words until one more word would cause the line to overflow.
  - "Justifying" a line means adding extra spaces between words so that each line has exactly the same length (60 characters).
- Justification must be done so that the space between words in a line is equal (or nearly equal).
- The last line of the output won't be justified.

Program: Text Formatting

- We assume that no word is longer than 20 characters, including any adjacent punctuation.
- If the program encounters a longer word, it must ignore all characters after the first 20, replacing them with a single asterisk.
- For example, the word

```
antidisestablishmentarianism
```

would be printed as

```
antidisestablishment*
```

Program: Text Formatting

- The program can’t write words one by one as they’re read.
- Instead, it will have to store them in a “line buffer” until there are enough to fill a line.

Program: Text Formatting

- The heart of the program will be a loop:

```
for (;;) {
  read word;
  if (can't read word) {
     write contents of line buffer without justification;
     terminate program;
  }
  if (word doesn't fit in line buffer) {
    write contents of line buffer with justification;
    clear line buffer;
  }
  add word to line buffer;
}
```

Program: Text Formatting

- The program will be split into three source files:
  - `word.c`: functions related to words
  - `line.c`: functions related to the line buffer
  - `justify.c`: contains the `main` function
- We'll also need two header files:
  - `word.h`: prototypes for the functions in `word.c`
  - `line.h`: prototypes for the functions in `line.c`
- `word.h` will contain the prototype for a function that reads a word.

word.h

```
#ifndef WORD_H
#define WORD_H

/***********************************************************
 * read_word: Reads the next word from the input and       *
 *            stores it in word. Makes word empty if no    *
 *            word could be read because of end-of-file.   *
 *            Truncates the word if its length exceeds     *
 *            len.                                         *

   **********************************************************
   */
void read_word(char *word, int len);

#endif
```

Program: Text Formatting

- The outline of the main loop reveals the need for functions that perform the following operations:
  - Write contents of line buffer without justification
  - Determine how many characters are left in line buffer
  - Write contents of line buffer with justification
  - Clear line buffer
  - Add word to line buffer
- We'll call these functions `flush_line`, `space_remaining`, `write_line`, `clear_line`, and `add_word`.

line.h

```
#ifndef LINE_H
#define LINE_H

/**********************************************************
 * clear_line: Clears the current line.                   *
 **********************************************************/
void clear_line(void);

/**********************************************************
 * add_word: Adds word to the end of the current line.    *
 *           If this is not the first word on the line,   *
 *           puts one space before word.                  *
 **********************************************************/
void add_word(const char *word);
```

```
/**********************************************************
 * space_remaining: Returns the number of characters left *
 *                  in the current line.                  *
 **********************************************************/
int space_remaining(void);

/**********************************************************
 * write_line: Writes the current line with               *
 *             justification.                             *
 **********************************************************/
void write_line(void);

/**********************************************************
 * flush_line: Writes the current line without            *
 *             justification. If the line is empty, does  *
 *             nothing.                                   *
 **********************************************************/
void flush_line(void);

#endif
```

Program: Text Formatting

- Before we write the `word.c` and `line.c` files, we can use the functions declared in `word.h` and `line.h` to write `justify.c`, the main program.
- Writing this file is mostly a matter of translating the original loop design into C.

justify.c

```
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;
```

```
  clear_line();
  for (;;) {
    read_word(word, MAX_WORD_LEN+1);
    word_len = strlen(word);
    if (word_len == 0) {
      flush_line();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX_WORD_LEN] = '*';
    if (word_len + 1 > space_remaining()) {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```

Program: Text Formatting

- `main` uses a trick to handle words that exceed 20 characters.
- When it calls `read_word`, `main` tells it to truncate any word that exceeds 21 characters.
- After `read_word` returns, `main` checks whether `word` contains a string that's longer than 20 characters.
- If so, the word must have been at least 21 characters long (before truncation), so `main` replaces its 21st character by an asterisk.

Program: Text Formatting

- The `word.h` header file has a prototype for only one function, `read_word`.
- `read_word` is easier to write if we add a small "helper" function, `read_char`.
- `read_char`'s job is to read a single character and, if it's a new-line character or tab, convert it to a space.
- Having `read_word` call `read_char` instead of `getchar` solves the problem of treating new-line characters and tabs as spaces.

word.c

```
#include <stdio.h>
#include "word.h"

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';
  return ch;
}
```

```
void read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
}
```

Program: Text Formatting

- `line.c` supplies definitions of the functions declared in `line.h`.
- `line.c` will also need variables to keep track of the state of the line buffer:
  - `line`: characters in the current line
  - `line_len`: number of characters in the current line
  - `num_words`: number of words in the current line

line.c

```
#include <stdio.h>
#include <string.h>
#include "line.h"
#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}
```

```
void add_word(const char *word)
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}
```

```
void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
}

void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```

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.

Makefiles

- Real makefiles aren't always easy to understand.
- There are numerous techniques that reduce the amount of redundancy in makefiles and make them easier to modify.
- These techniques greatly reduce the readability of makefiles.
- Alternatives to makefiles include the "project files" supported by some integrated development environments.

Errors During Linking

- Some errors that can't be detected during compilation will be found during linking.
- If the definition of a function or variable is missing from a program, the linker will be unable to resolve external references to it
- The result is a message such as "*undefined symbol*" or "*undefined reference.*"

Errors During Linking

- Common causes of errors during linking:
  - ***Misspellings.*** If the name of a variable or function is misspelled, the linker will report it as missing.
  - ***Missing files.*** If the linker can't find the functions that are in file `foo.c`, it may not know about the file.
  - ***Missing libraries.*** The linker may not be able to find all library functions used in the program.
- In UNIX, the `-lm` option may need to be specified when a program that uses `<math.h>` is linked.

Rebuilding a Program

- During the development of a program, it's rare that we'll need to compile all its files.
- To save time, the rebuilding process should recompile only those files that might be affected by the latest change.
- Assume that a program has been designed with a header file for each source file.
- To see how many files will need to be recompiled after a change, we need to consider two possibilities.

Rebuilding a Program

- If the change affects a single source file, only that file must be recompiled.
- Suppose that we decide to condense the `read_char` function in `word.c`:

```
int read_char(void)
{
  int ch = getchar();

  return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

- This modification doesn't affect `word.h`, so we need only recompile `word.c` and relink the program.

Rebuilding a Program

- The second possibility is that the change affects a header file.
- In that case, we should recompile all files that include the header file, since they could potentially be affected by the change.

Rebuilding a Program

- Suppose that we modify `read_word` so that it returns the length of the word that it reads.
- First, we change the prototype of `read_word` in `word.h`:

```
/**********************************************************
 * read_word: Reads the next word from the input and      *
 *            stores it in word. Makes word empty if no   *
 *            word could be read because of end-of-file.  *
 *            Truncates the word if its length exceeds    *
 *            len. Returns the number of characters       *
 *            stored.                                     *
 **********************************************************/
int read_word(char *word, int len);
```

Rebuilding a Program

- Next, we change the definition of `read_word`:

```
int read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
  return pos;
}
```

Rebuilding a Program

- Finally, we modify `justify.c` by removing the include of `<string.h>` and changing `main`:

```
int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();
  for (;;) {
    word_len = read_word(word, MAX_WORD_LEN+1);
    …
  }
}
```

Rebuilding a Program

- Once we've made these changes, we'll rebuild `justify` by recompiling `word.c` and `justify.c` and then relinking.
- A GCC command that rebuilds the program:

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

Rebuilding a Program

- One of the advantages of using makefiles is that rebuilding is handled automatically.
- By examining the date of each file, `make` can determine which files have changed since the program was last built.
- It then recompiles these files, together with all files that depend on them, either directly or indirectly.

Rebuilding a Program

- Suppose that we make the indicated changes to `word.h`, `word.c`, and `justify.c`.
- When the `justify` program is rebuilt, `make` will perform the following actions:
  1. Build `justify.o` by compiling `justify.c` (because `justify.c` and `word.h` were changed).
  2. Build `word.o` by compiling `word.c` (because `word.c` and `word.h` were changed).
  3. Build `justify` by linking `justify.o`, `word.o`, and `line.o` (because `justify.o` and `word.o` were changed).

Defining Macros Outside a Program

- C compilers usually provide some method of specifying the value of a macro at the time a program is compiled.
- This ability makes it easy to change the value of a macro without editing any of the program's files.
- It's especially valuable when programs are built automatically using makefiles.

Defining Macros Outside a Program

- Most compilers (including GCC) support the `-D` option, which allows the value of a macro to be specified on the command line:

  ```
  gcc -DDEBUG=1 foo.c
  ```

- In this example, the `DEBUG` macro is defined to have the value `1` in the program `foo.c`.
- If the `-D` option names a macro without specifying its value, the value is taken to be `1`.

Defining Macros Outside a Program

- Many compilers also support the `-U` option, which "undefines" a macro as if by using `#undef`.
- We can use `-U` to undefine a predefined macro or one that was defined earlier in the command line using `-D`.