15 Writing Large Programs

Around computers it is difficult to find the correct unit of time to measure progress. Some cathedrals took a century to complete. Can you imagine the grandeur and scope of a program that would take as long?

Although some C programs are small enough to be put in a single file, most aren't. Programs that consist of more than one file are the rule rather than the exception. In this chapter, we'll see that a typical program consists of several source files and usually some header files as well. Source files contain definitions of functions and external variables; header files contain information to be shared among source files. Section 15.1 discusses source files, while Section 15.2 covers header files. Section 15.3 describes how to divide a program into source files and header files. Section 15.4 then shows how to "build" (compile and link) a program that consists of more than one file, and how to "rebuild" a program after part of it has been changed.

15.1 Source Files

Up to this point, we've assumed that a C program consists of a single file. In fact, a 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.

For example, suppose that we want to write a simple calculator program that evaluates integer expressions entered in Reverse Polish notation (RPN), in which operators follow operands. If the user enters an expression such as

```
30 5 - 7 *
```

we want the program to print its value (175, in this case). Evaluating an RPN expression is easy if we have the program read the operands and operators, one by one, using a stack to keep track of intermediate results. If the program reads a

stacks ➤ 10.2

number, we'll have it push the number onto the stack. If it reads an operator, we'll have it 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. For example, the program will evaluate the expression `30 5 - 7 *` in the following way:

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.

After these steps, the stack will contain the value of the expression (175).

Turning this strategy into a program isn't hard. 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. 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`.

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—a great time-saver if the program is large and must be changed frequently (which is common during program development).
- Functions are more easily reused in other programs when grouped in separate source files. In our example, splitting off `stack.c` and `token.c` from the `main` function makes it simpler to reuse the stack functions and token functions in the future.

15.2 Header Files

When we divide a program into several source files, problems arise: How can a function in one file call a function that's defined in another file? How can a func-

tion 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—function prototypes, macro definitions, type definitions, and more—among any number of source files.

The `#include` directive tells the preprocessor to open a specified file and insert its contents into the current file. Thus, if we want several source files to have access to the same information, we'll put that information in a file and then use `#include` 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***); I'll discuss them in more detail later in this section. By convention, header files have the extension `.h`.

Note: The C standard uses the term "source file" to refer to all files written by the programmer, including both `.c` and `.h` files. I'll use "source file" to refer to `.c` files only.

The `#include` Directive

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

`#include` directive (form 1)

```
#include <filename>
```

The second form is used for all other header files, including any that we write:

`#include` directive (form 2)

```
#include "filename"
```

The difference between the two is a subtle one having to do with how the compiler locates the header file. Here are the rules that most compilers follow:

Q&A

- `#include <`*filename*`>`: Search the directory (or directories) in which system header files reside. (On UNIX systems, for example, system header files are usually kept in the directory `/usr/include`.)
- `#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*.

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 (and, of course, won't find it).

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. (That's fortunate, since `\c` and `\u`—which appear in the Windows example—would be treated as escape sequences in a string literal.)

portability tip

It's usually best not to include path or drive information in `#include` *directives. Such information makes it difficult to compile a program when it's transported to another machine or, worse, another operating system.*

For example, the following Windows `#include` directives specify drive and/or path information that may not always be valid:

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

The following directives are better; they don't mention specific drives, and paths are relative rather than absolute:

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

The `#include` directive has a third form that's used less often than the other two:

`#include` directive (form 3)

```
#include tokens
```

preprocessing tokens ➤ *14.3*

where *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, as the following example shows:

```
#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 (or, in the most extreme case, by *all* source files). These definitions should go into header files.

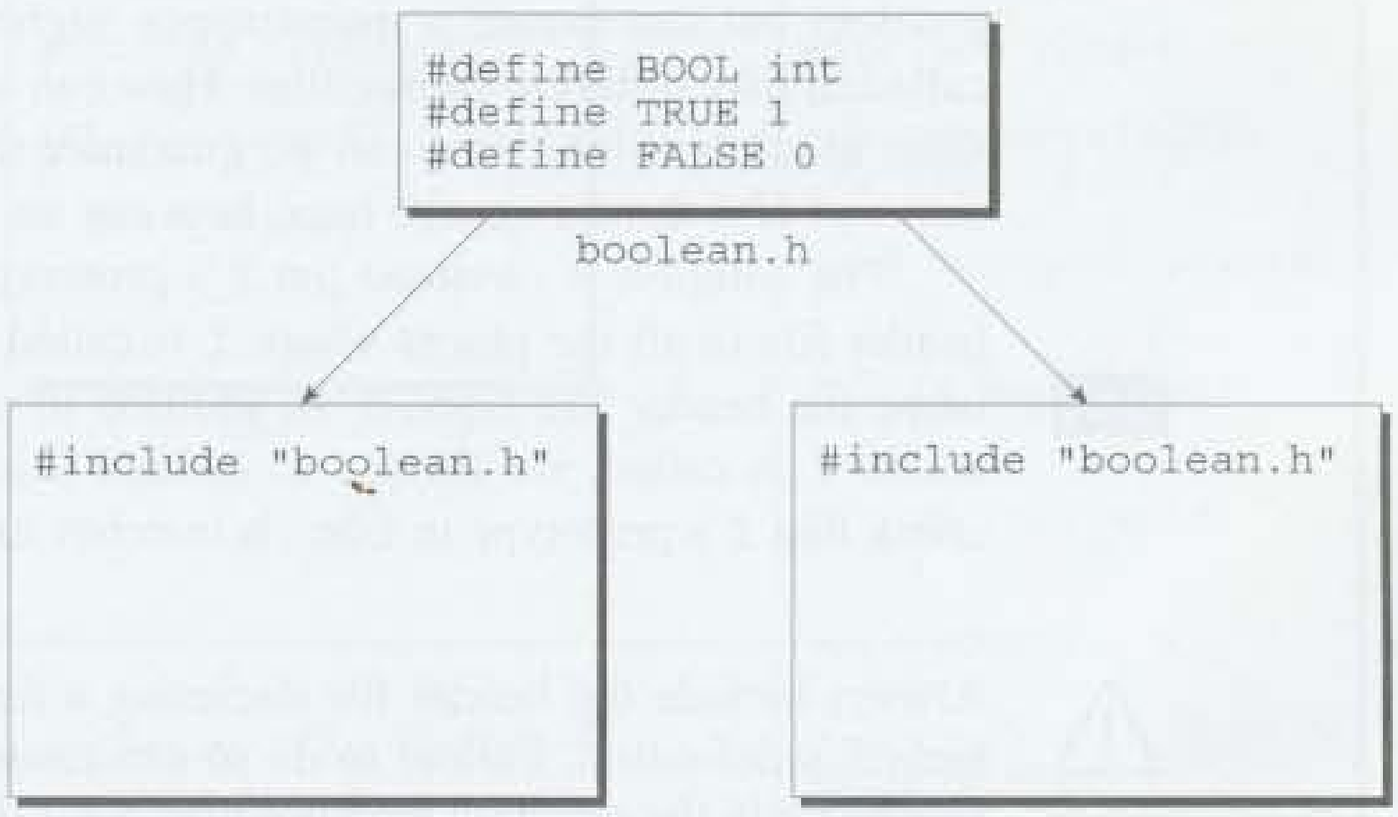
For example, suppose that we're writing a program that uses macros named `BOOL`, `TRUE`, and `FALSE`. (There's no need for these in C99, of course, because the `<stdbool.h>` header defines similar macros.) Instead of repeating the definitions of these macros in each source file that needs them, it makes more sense to put the definitions 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"
```

In the following figure, two files include `boolean.h`:



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;
```

Putting definitions of macros and types in header files has some clear advantages. First, we save time by not having to copy the definitions into the source files where they're needed. Second, the program becomes easier to modify. Changing the definition of a macro or type requires only that we edit a single header file; we don't have to modify the many source files in which the macro or type is used. Third, we don't have to worry about 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. Without a prototype to rely on, the compiler is forced to assume that `f`'s return type is `int` and that the number of parameters matches the number of arguments in the call of `f`. The arguments themselves are converted automatically to a kind of "standard form" by the default argument promotions. The compiler's assumptions may well be wrong, but it has no way to check them, since it compiles only one file at a time. If the assumptions are incorrect, the program probably won't work, and there won't be any clues as to why it doesn't. (For this reason, C99 prohibits calling a function for which the compiler has not yet seen a declaration or definition.)

default argument promotions ➤9.3

When calling a function `f` that's defined in another file, always make sure that the compiler has seen a prototype for `f` prior to the call.

Our first impulse is to declare `f` in the file where it's called. That solves the problem but can create a maintenance nightmare. Suppose that the function is called in fifty different source files. How can we ensure that `f`'s prototypes are the same in all the files? How can we guarantee that they match the definition of `f` in `foo.c`? If `f` should change later, how can we find all the files where it's used?

The solution is obvious: put `f`'s prototype in a header file, then include the header file in all the places where `f` is called. Since `f` is defined in `foo.c`, let's name the header file `foo.h`. In addition to including `foo.h` in the source files where `f` is called, we'll need to include it in `foo.c`, enabling the compiler to check that `f`'s prototype in `foo.h` matches its definition in `foo.c`.

Q&A

Always include the header file declaring a function `f` in the source file that contains `f`'s definition. Failure to do so can cause hard-to-find bugs, since calls of `f` elsewhere in the program may not match `f`'s definition.

If `foo.c` contains other functions, most of them should be declared in the same header file as `f`. After all, the other functions in `foo.c` are presumably related to `f`; any file that contains a call of `f` probably needs some of the other functions in `foo.c`. 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.

To illustrate the use of function prototypes in header files, let's return to the RPN calculator of Section 15.1. The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions. The following 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);
```

(To avoid complicating the example, `is_empty` and `is_full` will return `int` values instead of Boolean values.) 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`. The following figure shows `stack.h`, `stack.c`, and `calc.c`:

`stack.h`

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

`calc.c`

```
#include "stack.h"

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

`stack.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)
{ … }
```

Sharing Variable Declarations

external variables ➤ 10.2

External variables can be shared among files in much the same way functions are. To share a function, 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.

Up to this point, we haven't needed to distinguish between a variable's declaration and its definition. To declare a variable `i`, we've written

```
int i;              /* declares i and defines it as well */
```

which not only declares `i` to be a variable of type `int`, but defines `i` as well, by causing the compiler to set aside space for `i`. To declare `i` without defining it, we must put the keyword `extern` at the beginning of its declaration:

`extern` keyword ➤ 18.2

```
extern int i;   /* declares i without defining it */
```

`extern` informs the compiler that `i` is defined elsewhere in the program (most likely in a different source file), so there's no need to allocate space for it.

`extern` works with variables of all types. When we use it in the declaration of an array, we can omit the length of the array:

Q&A

```
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.

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. When this file is compiled, the compiler will allocate storage for `i`. 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. Because of the word `extern`, however, the compiler doesn't allocate additional storage for `i` each time one of the files is compiled.

When a variable is shared among files, we'll face a challenge similar to one that we had with shared functions: ensuring that all declarations of a variable agree with the definition of the variable.

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.

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.

Although sharing variables among files is a long-standing practice in the C world, it has significant disadvantages. In Section 19.2, we'll see what the problems are and learn how to design programs that don't need shared variables.

Nested Includes

A header file may itself contain `#include` directives. Although this practice may seem a bit odd, it can be quite useful in practice. Consider the `stack.h` file, which 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` instead of `int`, where `Bool` is the type that we defined earlier in this section:

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

Of course, 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. (In C99, we'd include `<stdbool.h>` instead of `boolean.h` and declare the return types of the two functions to be `bool` rather than `Bool`.)

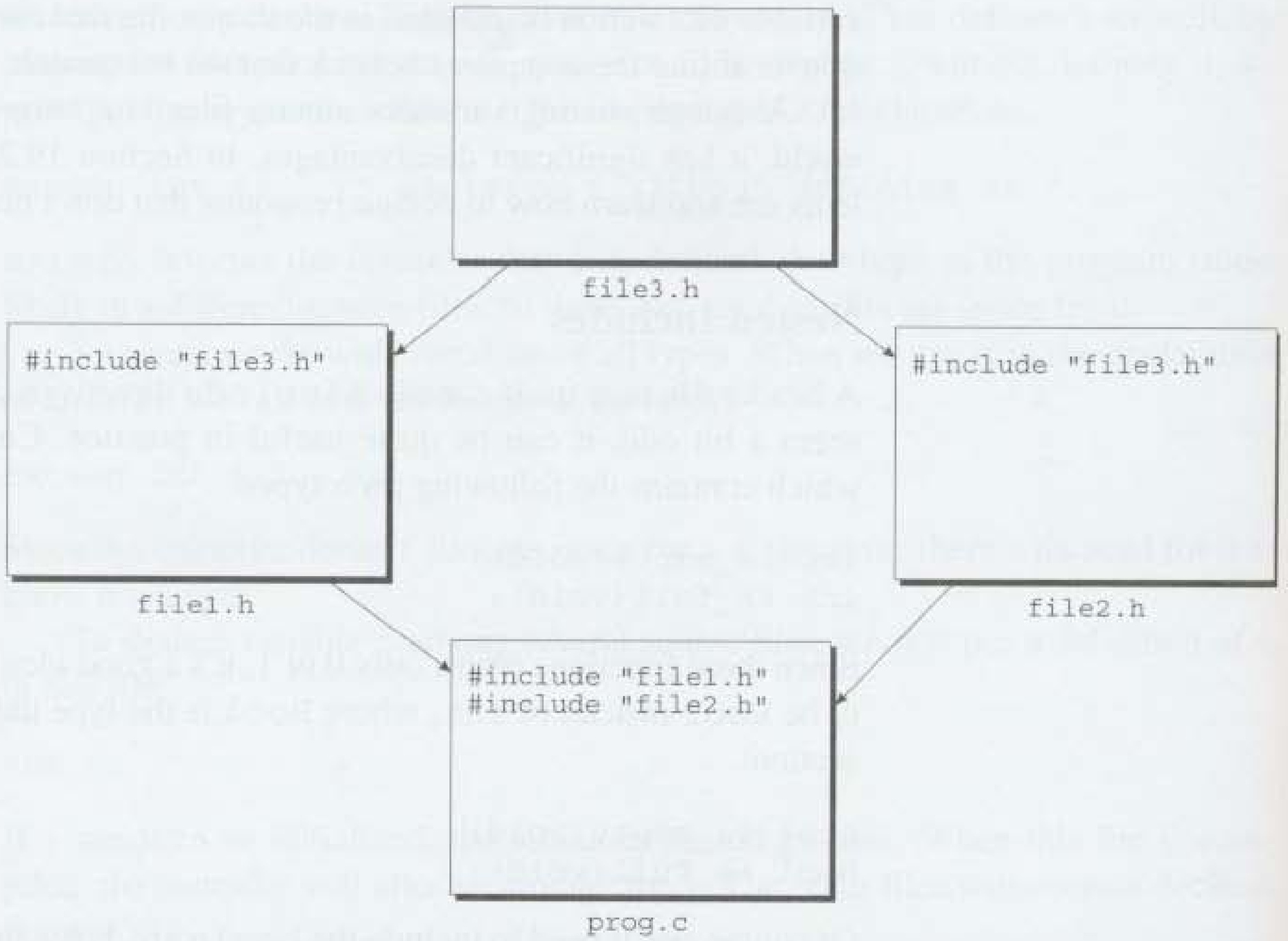
Traditionally, C programmers shun nested includes. (Early versions of C didn't allow them at all.) However, the bias against nested includes has largely faded away, in part because nested includes are common practice in C++.

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. For example, suppose that `file1.h` includes `file3.h`, `file2.h` includes `file3.h`, and `prog.c` includes both `file1.h` and `file2.h` (see the figure at the top of the next page). When `prog.c` is compiled, `file3.h` will be compiled twice.

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.

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.



To protect a header file, we'll enclose the contents of the file in an `#ifndef`-`#endif` pair. For example, the `boolean.h` file could be protected in the following way:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

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

#endif
```

When this file is included the first time, the `BOOLEAN_H` macro won't be defined, so the preprocessor will allow the lines between `#ifndef` and `#endif` to stay. But if the file should be included a second time, the preprocessor will remove the lines between `#ifndef` and `#endif`.

The name of the macro (`BOOLEAN_H`) doesn't really matter. However, making it 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` (identifiers can't contain periods), a name such as `BOOLEAN_H` is a good alternative.

`#error` Directives in Header Files

`#error` directives ➤ *14.5*

`#error` directives are often put in header files to check for conditions under which the header file shouldn't be included. For example, suppose that a header

file uses a feature that didn't exist prior to the original C89 standard. To prevent the header file from being used with older, nonstandard compilers, it could contain an `#ifndef` directive that tests for the existence of the `__STDC__` macro:

__STDC__ macro ➤ 14.3

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

15.3 Dividing a Program into Files

Let's now use what we know about header files and source files to develop a simple technique for dividing a program into files. We'll concentrate on functions, but the same principles apply to external variables as well. We'll assume that the program has already been designed; that is, we've decided what functions the program will need and how to arrange the functions into logically related groups. (We'll discuss program design in Chapter 19.)

Here's how we'll proceed. Each set of functions will go into a separate source file (let's use the name `foo.c` for one such file). In addition, we'll create a header file with the same name as the source file, but with the extension `.h` (`foo.h`, in our case). Into `foo.h`, we'll put prototypes for the functions defined in `foo.c`. (Functions that are designed for use only within `foo.c` need not—and should not—be declared in `foo.h`. The `read_char` function in our next program is an example.) We'll include `foo.h` in each source file that needs to call a function defined in `foo.c`. Moreover, we'll include `foo.h` in `foo.c` so that the compiler can check that the function prototypes in `foo.h` are consistent with the definitions in `foo.c`.

The `main` function will go in a file whose name matches the name of the program—if we want the program to be known as `bar`, then `main` should be in the file `bar.c`. 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

To illustrate the technique that we've just discussed, let's apply it to a small text-formatting program named `justify`. As sample input to `justify`, we'll use a file named `quote` that contains the following (poorly formatted) quotation from "The development of the C programming language" by Dennis M. Ritchie (in *History of Programming Languages II*, edited by T. J. Bergin, Jr., and R. G. Gibson, Jr., Addison-Wesley, Reading, Mass., 1996, pages 671–687):

```
   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
```

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***. When given the `quote` file as input, the `justify` program will produce the following output:

input redirection ➤22.1

```
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***:

output redirection ➤22.1

```
justify <quote >newquote
```

The output of `justify` will go into the file `newquote`.

In general, `justify`'s output should be identical to its input, except that extra spaces and blank lines are deleted, and lines are filled and justified. "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 as nearly equal as possible). The last line of the output won't be justified.

We'll assume that no word is longer than 20 characters. (A punctuation mark is considered part of the word to which it is adjacent.) That's a bit restrictive, of course, but once the program is written and debugged we can easily increase this limit to the point that it would virtually never be exceeded. 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*
```

Now that we understand what the program should do, it's time to think about a design. We'll start by observing that the program can't write the 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. After further reflection, we decide that the heart of the program will be a loop that goes something like this:

```
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;
}
```

Since we'll need functions that deal with words and functions that deal with the line buffer, let's split the program into three source files, putting all functions related to words in one file (`word.c`) and all functions related to the line buffer in another file (`line.c`). A third file (`justify.c`) will contain the `main` function. In addition to these files, we'll need two header files, `word.h` and `line.h`. The `word.h` file will contain prototypes for the functions in `word.c`; `line.h` will play a similar role for `line.c`.

By examining the main loop, we see that the only word-related function that we'll need is a `read_word` function. (If `read_word` can't read a word because it's reached the end of the input file, we'll have it signal the main loop by pretending to read an "empty" word.) Consequently, the `word.h` file is a small one:

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
```

Notice how the `WORD_H` macro protects `word.h` from being included more than once. Although `word.h` doesn't really need it, it's good practice to protect all header files in this way.

The `line.h` file won't be as short as `word.h`. Our 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`. Here's what the `line.h` header file will look like:

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
```

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 our 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);
  }
}
```

Including both `line.h` and `word.h` gives the compiler access to the function prototypes in both files as it compiles `justify.c`.

`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 that was read must have been at least 21 characters long (before truncation), so `main` replaces the word's 21st character by an asterisk.

Now it's time to write `word.c`. Although the `word.h` header file has a prototype for only one function, `read_word`, we can put additional functions in `word.c` if we need to. As it turns out, `read_word` is easier to write if we add a small "helper" function, `read_char`. We'll assign `read_char` the task of reading a single character and, if it's a new-line character or tab, converting 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.

Here's the `word.c` file:

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';
}
```

Before we discuss `read_word`, a couple of comments are in order concerning the use of `getchar` in the `read_char` function. First, `getchar` returns an `int` value instead of a `char` value; that's why the variable `ch` in `read_char` is declared to have type `int` and why the return type of `read_char` is `int`. Also, `getchar` returns the value `EOF` when it's unable to continue reading (usually because it has reached the end of the input file).

EOF macro ➤22.4

`read_word` consists of two loops. The first loop skips over spaces, stopping at the first nonblank character. (`EOF` isn't a blank, so the loop stops if it reaches the end of the input file.) The second loop reads characters until encountering a space or `EOF`. The body of the loop stores the characters in `word` until reaching the `len` limit. After that, the loop continues reading characters but doesn't store them. The final statement in `read_word` ends the word with a null character, thereby making it a string. If `read_word` encounters `EOF` before finding a nonblank character, `pos` will be 0 at the end, making `word` an empty string.

The only file left is `line.c`, which supplies definitions of the functions declared in the `line.h` file. `line.c` will also need variables to keep track of the state of the line buffer. One variable, `line`, will store the characters in the current line. Strictly speaking, `line` is the only variable we need. For speed and convenience, however, we'll use two other variables: `line_len` (the number of characters in the current line) and `num_words` (the number of words in the current line).

Here's the `line.c` file:

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);
}
```

Most of the functions in `line.c` are easy to write. The only tricky one is `write_line`, which writes a line with justification. `write_line` writes the characters in `line` one by one, pausing at the space between each pair of words to write additional spaces if needed. The number of additional spaces is stored in `spaces_to_insert`, which has the value `extra_spaces / (num_words - 1)`, where `extra_spaces` is initially the difference between the maximum line length and the actual line length. Since `extra_spaces` and `num_words` change after each word is printed, `spaces_to_insert` will change as well. If `extra_spaces` is 10 initially and `num_words` is 5, then the first word will be followed by 2 extra spaces, the second by 2, the third by 3, and the fourth by 3.

15.4 Building a Multiple-File Program

In Section 2.1, we examined the process of compiling and linking a program that fits into a single file. Let's expand that discussion to cover multiple-file programs. Building a large program requires the same basic steps as building a small one:

- ***Compiling.*** 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.
- ***Linking.*** 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.)

Most compilers allow us to build a program in a single step. With the GCC compiler, for example, we'd use the following command to build the `justify` program of Section 15.3:

```
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

Putting the names of all the source files on the command line quickly gets tedious. Worse still, we could waste a lot of time when rebuilding a program if we recompile all source files, not just the ones that were affected by our most recent changes.

To make it easier to build large programs, UNIX originated the concept of the ***makefile,*** a file containing the information necessary to build a program. 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`.

Here's a UNIX makefile for the `justify` program. The makefile uses GCC for compilation and linking:

```
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
```

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. Let's look at the first two rules; the last two are similar.

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 one of these three files has changed since the program was last built, then `justify` needs to be rebuilt. The command on the following line shows how the rebuilding is to be done (by using the `gcc` command to link the three object files).

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 reason for mentioning `word.h` and `line.h` is that `justify.c` includes both these files, so it's potentially affected by a change to either one.) The next line shows how to update `justify.o` (by recompiling `justify.c`). The `-c` option tells the compiler to compile `justify.c` into an object file but not attempt to link it.

Q&A 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.

If you want to give `make` a try, here are a few details you'll need to know:

- Each command in a makefile must be preceded by a tab character, not a series of spaces. (In our example, the commands appear to be indented eight spaces, but it's actually a single tab character.)
- 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.
- To invoke `make`, use the command

  `make` *target*

  where *target* is one of the targets listed in the makefile. To build the `justify` executable using our makefile, we would use the command

  ```
  make justify
  ```

- If no target is specified when `make` is invoked, it will build the target of the first rule. For example, the command

  ```
  make
  ```

  will build the `justify` executable, since `justify` is the first target in our makefile. Except for this special property of the first rule, the order of rules in a makefile is arbitrary.

`make` is complicated enough that entire books have been written about it, so we won't attempt to delve further into its intricacies. Let's just say that real makefiles aren't usually as easy to understand as our example. There are numerous techniques that reduce the amount of redundancy in makefiles and make them easier to modify; at the same time, though, these techniques greatly reduce their readability.

Not everyone uses makefiles, by the way. Other program maintenance tools are also popular, including 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. In particular, if the definition of a function or variable is missing from a program, the linker will be unable to resolve external references to it, causing a message such as *"undefined symbol"* or *"undefined reference."*

Errors detected by the linker are usually easy to fix. Here are some of the most common causes:

- ***Misspellings.*** If the name of a variable or function is misspelled, the linker will report it as missing. For example, if the function `read_char` is defined

in the program but called as `read_cahr`, the linker will report that `read_cahr` is missing.

- ***Missing files.*** If the linker can't find the functions that are in file `foo.c`, it may not know about the file. Check the makefile or project file to make sure that `foo.c` is listed there.
- ***Missing libraries.*** The linker may not be able to find all library functions used in the program. A classic example occurs in UNIX programs that use the `<math.h>` header. Simply including the header in a program may not be enough; many versions of UNIX require that the `-lm` option be specified when the program is linked, causing the linker to search a system file that contains compiled versions of the `<math.h>` functions. Failing to use this option may cause "undefined reference" messages during linking.

Rebuilding a Program

During the development of a program, it's rare that we'll need to compile all its files. Most of the time, we'll test the program, make a change, then build the program again. To save time, the rebuilding process should recompile only those files that might be affected by the latest change.

Let's assume that we've designed our program in the way outlined in Section 15.3, 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.

The first possibility is that the change affects a single source file. In that case, only that file must be recompiled. (After that, the entire program will need to be relinked, of course.) Consider the `justify` program. Suppose that we decide to condense the `read_char` function in `word.c` (changes are marked in **bold**):

```
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.

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. (Some of them might not be, but it pays to be conservative.)

As an example, consider the `read_word` function in the `justify` program. Notice that `main` calls `strlen` immediately after calling `read_word`, in order to determine the length of the word that was just read. Since `read_word` already knows the length of the word (`read_word`'s `pos` variable keeps track of the length), it seems silly to use `strlen`. Modifying `read_word` to return the word's length is easy. 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);
```

Of course, we're careful to change the comment that accompanies `read_word`. Next, we change the definition of `read_word` in `word.c`:

```
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;
}
```

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

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

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

Once we've made these changes, we'll rebuild the `justify` program by recompiling `word.c` and `justify.c` and then relinking. There's no need to recompile `line.c`, which doesn't include `word.h` and therefore won't be affected by changes to it. With the GCC compiler, we could use the following command to rebuild the program:

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

Note the mention of `line.o` instead of `line.c`.

One of the advantages of using makefiles is that rebuilding is handled automatically. By examining the date of each file, the `make` utility 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. For example, if we make the indicated changes to `word.h`, `word.c`, and `justify.c` and then rebuild the `justify` program, `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.

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`, just as if the line

```
#define DEBUG 1
```

appeared at the beginning of `foo.c`. If the `-D` option names a macro without specifying its value, the value is taken to be 1.

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`.

predefined macros ➤ 14.3

Q & A

Q: **You don't have any examples that use the `#include` directive to include a source file. What would happen if we were to do this?**

A: That's not a good practice, although it's not illegal. Here's an example of the kind of trouble you can get into. Suppose that `foo.c` defines a function `f` that we'll need in `bar.c` and `baz.c`, so we put the directive

```
#include "foo.c"
```

in both `bar.c` and `baz.c`. Each of these files will compile nicely. The problem comes later, when the linker discovers two copies of the object code for `f`. Of course, we would have gotten away with including `foo.c` if only `bar.c` had included it, not `baz.c` as well. To avoid problems, it's best to use `#include` only with header files, not source files.

Q: **What are the exact search rules for the `#include` directive? [p. 351]**

A: That depends on your compiler. The C standard is deliberately vague in its description of `#include`. If the file name is enclosed in *brackets*, the preprocessor looks in a "sequence of implementation-defined places," as the standard obliquely puts it. If the file name is enclosed in *quotation marks*, the file "is searched for in an implementation-defined manner" and, if not found, then searched as if its name had been enclosed in brackets. The reason for this waffling is simple: not all operating systems have hierarchical (tree-like) file systems.

To make matters even more interesting, the standard doesn't require that names enclosed in brackets be file names at all, leaving open the possibility that `#include` directives using `<>` are handled entirely within the compiler.

Q: **I don't understand why each source file needs its own header file. Why not have one big header file containing macro definitions, type definitions, and function prototypes? By including this file, each source file would have access to all the shared information it needs. [p. 354]**

A: The "one big header file" approach certainly works; a number of programmers use it. And it does have an advantage: with only one header file, there are fewer files to manage. For large programs, however, the disadvantages of this approach tend to outweigh its advantages.

Using a single header file provides no useful information to someone reading the program later. With multiple header files, the reader can quickly see what other parts of the program are used by a particular source file.

But that's not all. Since each source file depends on the big header file, changing it will cause all source files to be recompiled—a significant drawback in a large program. To make matters worse, the header file will probably change frequently because of the large amount of information it contains.

Q: The chapter says that a shared array should be declared as follows:

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

Since arrays and pointers are closely related, would it be legal to write

```
extern int *a;
```

instead? [p. 356]

A: No. When used in expressions, arrays "decay" into pointers. (We've noticed this behavior when an array name is used as an argument in a function call.) In variable declarations, however, arrays and pointers are distinct types.

Q: Does it hurt if a source file includes headers that it doesn't really need?

A: Not unless the header has a declaration or definition that conflicts with one in the source file. Otherwise, the worst that can happen is a minor increase in the time it takes to compile the source file.

Q: I needed to call a function in the file `foo.c`, so I included the matching header file, `foo.h`. My program compiled, but it won't link. Why?

A: Compilation and linking are completely separate in C. Header files exist to provide information to the compiler, not the linker. If you want to call a function in `foo.c`, then you have to make sure that `foo.c` is compiled and that the linker is aware that it must search the object file for `foo.c` to find the function. Usually this means naming `foo.c` in the program's makefile or project file.

Q: If my program calls a function in `<stdio.h>`, does that mean that all functions in `<stdio.h>` will be linked with the program?

A: No. Including `<stdio.h>` (or any other header) has no effect on linking. In any event, most linkers will link only functions that your program actually needs.

Q: Where can I get the `make` utility? [p. 367]

A: `make` is a standard UNIX utility. The GNU version, known as GNU Make, is included in most Linux distributions. It's also available directly from the Free Software Foundation (*www.gnu.org/software/make/*).

Exercises

Section 15.1

1. Section 15.1 listed several advantages of dividing a program into multiple source files.
   (a) Describe several other advantages.
   (b) Describe some disadvantages.

Section 15.2

2. Which of the following should *not* be put in a header file? Why not?
   (a) Function prototypes
   (b) Function definitions

(c) Macro definitions
(d) Type definitions

3. We saw that writing `#include <`*file*`>` instead of `#include "`*file*`"` may not work if *file* is one that we've written. Would there be any problem with writing `#include "`*file*`"` instead of `#include <`*file*`>` if *file* is a system header?

4. Assume that `debug.h` is a header file with the following contents:

```
#ifdef DEBUG
#define PRINT_DEBUG(n) printf("Value of " #n ": %d\n", n)
#else
#define PRINT_DEBUG(n)
#endif
```

Let `testdebug.c` be the following source file:

```
#include <stdio.h>

#define DEBUG
#include "debug.h"

int main(void)
{
  int i = 1, j = 2, k = 3;

#ifdef DEBUG
  printf("Output if DEBUG is defined:\n");
#else
  printf("Output if DEBUG is not defined:\n");
#endif

  PRINT_DEBUG(i);
  PRINT_DEBUG(j);
  PRINT_DEBUG(k);
  PRINT_DEBUG(i + j);
  PRINT_DEBUG(2 * i + j - k);

  return 0;
}
```

(a) What is the output when the program is executed?
(b) What is the output if the `#define` directive is removed from `testdebug.c`?
(c) Explain why the output is different in parts (a) and (b).
(d) Is it necessary for the `DEBUG` macro to be defined *before* `debug.h` is included in order for `PRINT_DEBUG` to have the desired effect? Justify your answer.

Section 15.4

5. Suppose that a program consists of three source files—`main.c`, `f1.c`, and `f2.c`—plus two header files, `f1.h` and `f2.h`. All three source files include `f1.h`, but only `f1.c` and `f2.c` include `f2.h`. Write a makefile for this program, assuming that the compiler is `gcc` and that the executable file is to be named `demo`.

W 6. The following questions refer to the program described in Exercise 5.

(a) Which files need to be compiled when the program is built for the first time?
(b) If `f1.c` is changed after the program has been built, which files need to be recompiled?
(c) If `f1.h` is changed after the program has been built, which files need to be recompiled?
(d) If `f2.h` is changed after the program has been built, which files need to be recompiled?

Programming Projects

1. The `justify` program of Section 15.3 justifies lines by inserting extra spaces between words. The way the `write_line` function currently works, the words closer to the end of a line tend to have slightly wider gaps between them than the words at the beginning. (For example, the words closer to the end might have three spaces between them, while the words closer to the beginning might be separated by only two spaces.) Improve the program by having `write_line` alternate between putting the larger gaps at the end of the line and putting them at the beginning of the line.

2. Modify the `justify` program of Section 15.3 by having the `read_word` function (instead of `main`) store the * character at the end of a word that's been truncated.

3. Modify the `qsort.c` program of Section 9.6 so that the `quicksort` and `split` functions are in a separate file named `quicksort.c`. Create a header file named `quicksort.h` that contains prototypes for the two functions and have both `qsort.c` and `quicksort.c` include this file.

4. Modify the `remind.c` program of Section 13.5 so that the `read_line` function is in a separate file named `readline.c`. Create a header file named `readline.h` that contains a prototype for the function and have both `remind.c` and `readline.c` include this file.

5. Modify Programming Project 6 from Chapter 10 so that it has separate `stack.h` and `stack.c` files, as described in Section 15.2.

16 Structures, Unions, and Enumerations

Functions delay binding: data structures induce binding.
Moral: Structure data late in the programming process.

This chapter introduces three new types: structures, unions, and enumerations. A structure is a collection of values (members), possibly of different types. A union is similar to a structure, except that its members share the same storage; as a result, a union can store one member at a time, but not all members simultaneously. An enumeration is an integer type whose values are named by the programmer.

Of these three types, structures are by far the most important, so I'll devote most of the chapter to them. Section 16.1 shows how to declare structure variables and perform basic operations on them. Section 16.2 then explains how to define structure types, which—among other things—allow us to write functions that accept structure arguments or return structures. Section 16.3 explores how arrays and structures can be nested. The last two sections are devoted to unions (Section 16.4) and enumerations (Section 16.5).

16.1 Structure Variables

The only data structure we've covered so far is the array. Arrays have two important properties. First, all elements of an array have the same type. Second, to select an array element, we specify its position (as an integer subscript).

The properties of a ***structure*** are quite different from those of an array. The elements of a structure (its ***members***, in C parlance) aren't required to have the same type. Furthermore, the members of a structure have names; to select a particular member, we specify its name, not its position.

Structures may sound familiar, since most programming languages provide a similar feature. In some languages, structures are called ***records***, and members are known as ***fields***.

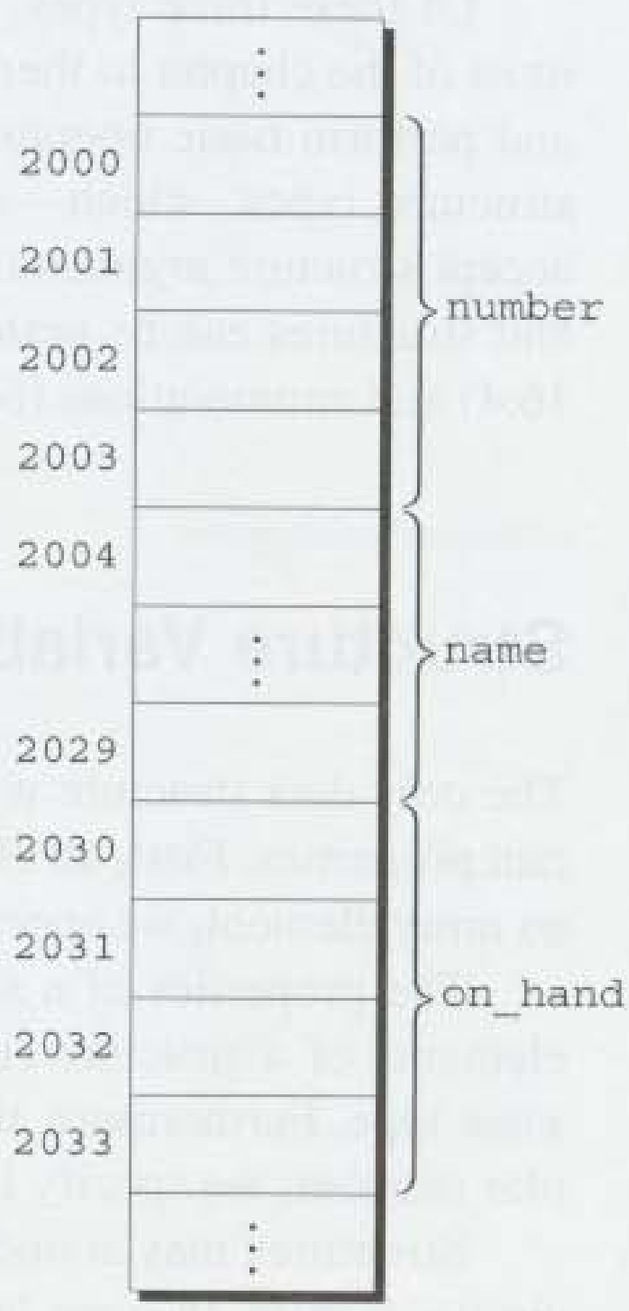
Declaring Structure Variables

When we need to store a collection of related data items, a structure is a logical choice. For example, suppose that we need to keep track of parts in a warehouse. The information that we'll need to store for each part might include a part number (an integer), a part name (a string of characters), and the number of parts on hand (an integer). To create variables that can store all three items of data, we might use a declaration such as the following:

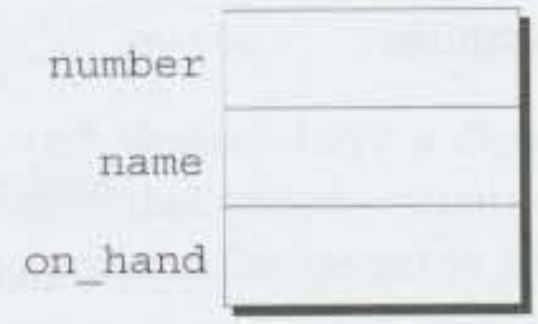
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Each structure variable has three members: `number` (the part number), `name` (the name of the part), and `on_hand` (the quantity on hand). Notice that this declaration has the same form as other variable declarations in C: `struct { ... }` specifies a type, while `part1` and `part2` are variables of that type.

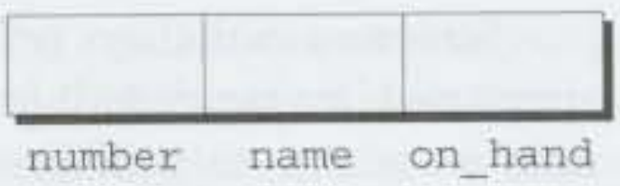
The members of a structure are stored in memory in the order in which they're declared. In order to show what the `part1` variable looks like in memory, let's assume that (1) `part1` is located at address 2000, (2) integers occupy four bytes, (3) `NAME_LEN` has the value 25, and (4) there are no gaps between the members. With these assumptions, `part1` will have the following appearance:



Usually it's not necessary to draw structures in such detail. I'll normally show them more abstractly, as a series of boxes:



I may sometimes draw the boxes horizontally instead of vertically:



Member values will go in the boxes later; for now, I've left them empty.

Each structure represents a new scope; any names declared in that scope won't conflict with other names in a program. (In C terminology, we say that each structure has a separate ***name space*** for its members.) For example, the following declarations can appear in the same program:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

The `number` and `name` members in the `part1` and `part2` structures don't conflict with the `number` and `name` members in `employee1` and `employee2`.

Initializing Structure Variables

Like an array, a structure variable may be initialized at the time it's declared. To initialize a structure, we prepare a list of values to be stored in the structure and enclose it in braces:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

The values in the initializer must appear in the same order as the members of the structure. In our example, the `number` member of `part1` will be 528, the `name` member will be `"Disk drive"`, and so on. Here's how `part1` will look after initialization:

number	528
name	Disk drive
on_hand	10

Structure initializers follow rules similar to those for array initializers. Expressions used in a structure initializer must be constant; for example, we couldn't have used a variable to initialize `part1`'s `on_hand` member. (This restriction is relaxed in C99, as we'll see in Section 18.5.) An initializer can have fewer members than the structure it's initializing; as with arrays, any "leftover" members are given 0 as their initial value. In particular, the bytes in a leftover character array will be zero, making it represent the empty string.

C99

Designated Initializers

C99's designated initializers, which were discussed in Section 8.1 in the context of arrays, can also be used with structures. Consider the initializer for `part1` shown in the previous example:

```
{528, "Disk drive", 10}
```

A designated initializer would look similar, but with each value labeled by the name of the member that it initializes:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

The combination of the period and the member name is called a ***designator.*** (Designators for array elements have a different form.)

Designated initializers have several advantages. For one, they're easier to read and check for correctness, because the reader can clearly see the correspondence between the members of the structure and the values listed in the initializer. Another is that the values in the initializer don't have to be placed in the same order that the members are listed in the structure. Our example initializer could be written as follows:

```
{.on_hand = 10, .name = "Disk drive", .number = 528}
```

Since the order doesn't matter, the programmer doesn't have to remember the order in which the members were originally declared. Moreover, the order of the members can be changed in the future without affecting designated initializers.

Not all values listed in a designated initializer need be prefixed by a designator. (This is true for arrays as well, as we saw in Section 8.1.) Consider the following example:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

The value `"Disk drive"` doesn't have a designator, so the compiler assumes that it initializes the member that follows `number` in the structure. Any members that the initializer fails to account for are set to zero.

Operations on Structures

Since the most common array operation is subscripting—selecting an element by position—it's not surprising that the most common operation on a structure is selecting one of its members. Structure members are accessed by name, though, not by position.

To access a member within a structure, we write the name of the structure first, then a period, then the name of the member. For example, the following statements will display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

lvalues ➤4.2

The members of a structure are lvalues, so they can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;          /* changes part1's part number */
part1.on_hand++;    /* increments part1's quantity on hand */
```

table of operators ➤Appendix A

The period that we use to access a structure member is actually a C operator. It has the same precedence as the postfix ++ and -- operators, so it takes precedence over nearly all other operators. Consider the following example:

```
scanf("%d", &part1.on_hand);
```

The expression `&part1.on_hand` contains two operators (`&` and `.`). The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`, as we wished.

The other major structure operation is assignment:

```
part2 = part1;
```

The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Since arrays can't be copied using the = operator, it comes as something of a surprise to discover that structures can. It's even more surprising when you consider that an array embedded within a structure is copied when the enclosing structure is copied. Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;

a1 = a2;   /* legal, since a1 and a2 are structures */
```

The = operator can be used only with structures of ***compatible*** types. Two structures declared at the same time (as `part1` and `part2` were) are compatible. As we'll see in the next section, structures declared using the same "structure tag" or the same type name are also compatible.

Other than assignment, C provides no operations on entire structures. In particular, we can't use the == and != operators to test whether two structures are equal or not equal.

Q&A

16.2 Structure Types

Although the previous section showed how to declare structure *variables*, it failed to discuss an important issue: naming structure *types*. Suppose that a program needs to declare several structure variables with identical members. If all the variables can be declared at one time, there's no problem. But if we need to declare the variables at different points in the program, then life becomes more difficult. If we write

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1;
```

in one place and

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part2;
```

in another, we'll quickly run into problems. Repeating the structure information will bloat the program. Changing the program later will be risky, since we can't easily guarantee that the declarations will remain consistent.

But those aren't the biggest problems. According to the rules of C, `part1` and `part2` don't have compatible types. As a result, `part1` can't be assigned to `part2`, and vice versa. Also, since we don't have a name for the type of `part1` or `part2`, we can't use them as arguments in function calls.

To avoid these difficulties, we need to be able to define a name that represents a *type* of structure, not a particular structure *variable*. As it turns out, C provides two ways to name structures: we can either declare a "structure tag" or use `typedef` to define a type name.

Q&A

type definitions ➤7.5

Declaring a Structure Tag

A ***structure tag*** is a name used to identify a particular kind of structure. The following example declares a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

Notice the semicolon that follows the right brace—it must be present to terminate the declaration.

Accidentally omitting the semicolon at the end of a structure declaration can cause surprising errors. Consider the following example:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
}                /*** WRONG: semicolon missing ***/

f(void)
{
  ...
  return 0;     /* error detected at this line */
}
```

The programmer failed to specify the return type of the function `f` (a bit of sloppy programming). Since the preceding structure declaration wasn't terminated properly, the compiler assumes that `f` returns a value of type `struct part`. The error won't be detected until the compiler reaches the first `return` statement in the function. The result: a cryptic error message.

Once we've created the `part` tag, we can use it to declare variables:

```
struct part part1, part2;
```

Unfortunately, we can't abbreviate this declaration by dropping the word `struct`:

```
part part1, part2;   /*** WRONG ***/
```

`part` isn't a type name; without the word `struct`, it is meaningless.

Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program. It would be perfectly legal (although more than a little confusing) to have a variable named `part`.

Incidentally, the declaration of a structure *tag* can be combined with the declaration of structure *variables*:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Here, we've declared a structure tag named `part` (making it possible to use `part` later to declare more variables) as well as variables named `part1` and `part2`.

All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;   /* legal; both parts have the same type */
```

Defining a Structure Type

As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name. For example, we could define a type named `Part` in the following way:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

Note that the name of the type, `Part`, must come at the end, not after the word `struct`.

We can use `Part` in the same way as the built-in types. For example, we might use it to declare variables:

```
Part part1, part2;
```

Since `Part` is a `typedef` name, we're not allowed to write `struct Part`. All `Part` variables, regardless of where they're declared, are compatible.

When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`. However, as we'll see later, declaring a structure tag is mandatory when the structure is to be used in a linked list. I'll use structure tags rather than `typedef` names in most of my examples.

Q&A

linked lists ➤ 17.5

Structures as Arguments and Return Values

Functions may have structures as arguments and return values. Let's look at two examples. Our first function, when given a `part` structure as its argument, prints the structure's members:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
```

```
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

Here's how `print_part` might be called:

```
print_part(part1);
```

Our second function returns a `part` structure that it constructs from its arguments:

```
struct part build_part(int number, const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

Notice that it's legal for `build_part`'s parameters to have names that match the members of the `part` structure, since the structure has its own name space. Here's how `build_part` might be called:

```
part1 = build_part(528, "Disk drive", 10);
```

Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure. As a result, these operations impose a fair amount of overhead on a program, especially if the structure is large. To avoid this overhead, it's sometimes advisable to pass a *pointer* to a structure instead of passing the structure itself. Similarly, we might have a function return a pointer to a structure instead of returning an actual structure. Section 17.5 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.

There are other reasons to avoid copying structures besides efficiency. For example, the `<stdio.h>` header defines a type named `FILE`, which is typically a structure. Each `FILE` structure stores information about the state of an open file and therefore must be unique in a program. Every function in `<stdio.h>` that opens a file returns a pointer to a `FILE` structure, and every function that performs an operation on an open file requires a `FILE` pointer as an argument.

`FILE` type ➤22.1

On occasion, we may want to initialize a structure variable inside a function to match another structure, possibly supplied as a parameter to the function. In the following example, the initializer for `part2` is the parameter passed to the `f` function:

```
void f(struct part part1)
{
  struct part part2 = part1;
  ...
}
```

automatic storage duration ➤ *10.1*

C permits initializers of this kind, provided that the structure we're initializing (`part2`, in this case) has automatic storage duration (it's local to a function and hasn't been declared `static`). The initializer can be any expression of the proper type, including a function call that returns a structure.

C99 Compound Literals

Section 9.3 introduced the C99 feature known as the ***compound literal.*** In that section, compound literals were used to create unnamed arrays, usually for the purpose of passing the array to a function. A compound literal can also be used to create a structure "on the fly," without first storing it in a variable. The resulting structure can be passed as a parameter, returned by a function, or assigned to a variable. Let's look at a couple of examples.

First, we can use a compound literal to create a structure that will be passed to a function. For example, we could call the `print_part` function as follows:

```
print_part((struct part) {528, "Disk drive", 10});
```

The compound literal (shown in **bold**) creates a `part` structure containing the members 528, `"Disk drive"`, and 10, in that order. This structure is then passed to `print_part`, which displays it.

Here's how a compound literal might be assigned to a variable:

```
part1 = (struct part) {528, "Disk drive", 10};
```

This statement resembles a declaration containing an initializer, but it's not the same—initializers can appear only in declarations, not in statements such as this one.

In general, a compound literal consists of a type name within parentheses, followed by a set of values enclosed by braces. In the case of a compound literal that represents a structure, the type name can be a structure tag preceded by the word `struct`—as in our examples—or a `typedef` name. A compound literal may contain designators, just like a designated initializer:

```
print_part((struct part) {.on_hand = 10,
                          .name = "Disk drive",
                          .number = 528});
```

A compound literal may fail to provide full initialization, in which case any uninitialized members default to zero.

16.3 Nested Arrays and Structures

Structures and arrays can be combined without restriction. Arrays may have structures as their elements, and structures may contain arrays and structures as members. We've already seen an example of an array nested inside a structure (the

name member of the part structure). Let's explore the other possibilities: structures whose members are structures and arrays whose elements are structures.

Nested Structures

Nesting one kind of structure inside another is often useful. For example, suppose that we've declared the following structure, which can store a person's first name, middle initial, and last name:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

We can use the person_name structure as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

Accessing student1's first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

One advantage of making name a structure (instead of having first, middle_initial, and last be members of the student structure) is that we can more easily treat names as units of data. For example, if we were to write a function that displays a name, we could pass it just one argument—a person_name structure—instead of three arguments:

```
display_name(student1.name);
```

Likewise, copying the information from a person_name structure to the name member of a student structure would take one assignment instead of three:

```
struct person_name new_name;
...
student1.name = new_name;
```

Arrays of Structures

One of the most common combinations of arrays and structures is an array whose elements are structures. An array of this kind can serve as a simple database. For example, the following array of part structures is capable of storing information about 100 parts:

```
struct part inventory[100];
```

To access one of the parts in the array, we'd use subscripting. To print the part stored in position `i`, for example, we could write

```
print_part(inventory[i]);
```

Accessing a member within a `part` structure requires a combination of subscripting and member selection. To assign 883 to the `number` member of `inventory[i]`, we could write

```
inventory[i].number = 883;
```

Accessing a single character in a part name requires subscripting (to select a particular part), followed by selection (to select the `name` member), followed by subscripting (to select a character within the part name). To change the name stored in `inventory[i]` to an empty string, we could write

```
inventory[i].name[0] = '\0';
```

Initializing an Array of Structures

Initializing an array of structures is done in much the same way as initializing a multidimensional array. Each structure has its own brace-enclosed initializer; the initializer for the array simply wraps another set of braces around the structure initializers.

One reason for initializing an array of structures is that we're planning to treat it as a database of information that won't change during program execution. For example, suppose that we're working on a program that will need access to the country codes used when making international telephone calls. First, we'll set up a structure that can store the name of a country along with its code:

```
struct dialing_code {
  char *country;
  int code;
};
```

Note that `country` is a pointer, not an array of characters. That could be a problem if we were planning to use `dialing_code` structures as variables, but we're not. When we initialize a `dialing_code` structure, `country` will end up pointing to a string literal.

Next, we'll declare an array of these structures and initialize it to contain the codes for some of the world's most populous nations:

```
const struct dialing_code country_codes[] =
  {{"Argentina",              54}, {"Bangladesh",       880},
   {"Brazil",                 55}, {"Burma (Myanmar)",   95},
   {"China",                  86}, {"Colombia",          57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",               20},
   {"Ethiopia",               251}, {"France",           33},
   {"Germany",                49}, {"India",             91},
```

```
{"Indonesia",      62}, {"Iran",          98},
{"Italy",          39}, {"Japan",         81},
{"Mexico",         52}, {"Nigeria",      234},
{"Pakistan",       92}, {"Philippines",   63},
{"Poland",         48}, {"Russia",         7},
{"South Africa",   27}, {"South Korea",   82},
{"Spain",          34}, {"Sudan",        249},
{"Thailand",       66}, {"Turkey",        90},
{"Ukraine",       380}, {"United Kingdom", 44},
{"United States",   1}, {"Vietnam",       84}};
```

The inner braces around each structure value are optional. As a matter of style, however, I prefer not to omit them.

C99 Because arrays of structures (and structures containing arrays) are so common, C99's designated initializers allow an item to have more than one designator. Suppose that we want to initialize the `inventory` array to contain a single part. The part number is 528 and the quantity on hand is 10, but the name is to be left empty for now:

```
struct part inventory[100] =
  {[0].number = 528, [0].on_hand = 10, [0].name[0] = '\0'};
```

The first two items in the list use two designators (one to select array element 0—a `part` structure—and one to select a member within the structure). The last item uses three designators: one to select an array element, one to select the `name` member within that element, and one to select element 0 of `name`.

PROGRAM Maintaining a Parts Database

To illustrate how nested arrays and structures are used in practice, we'll now develop a fairly long program that maintains a database of information about parts stored in a warehouse. The program is built around an array of structures, with each structure containing information—part number, name, and quantity—about one part. Our program will support the following operations:

- ***Add a new part number, part name, and initial quantity on hand.*** The program must print an error message if the part is already in the database or if the database is full.
- ***Given a part number, print the name of the part and the current quantity on hand.*** The program must print an error message if the part number isn't in the database.
- ***Given a part number, change the quantity on hand.*** The program must print an error message if the part number isn't in the database.
- ***Print a table showing all information in the database.*** Parts must be displayed in the order in which they were entered.
- ***Terminate program execution.***

We'll use the codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) to represent these operations. A session with the program might look like this:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10

Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name                      Quantity on Hand
    528       Disk drive                            8
    914       Printer cable                         5

Enter operation code: q
```

The program will store information about each part in a structure. We'll limit the size of the database to 100 parts, making it possible to store the structures in an array, which I'll call `inventory`. (If this limit proves to be too small, we can always change it later.) To keep track of the number of parts currently stored in the array, we'll use a variable named `num_parts`.

Since this program is menu-driven, it's fairly easy to sketch the main loop:

```
for (;;) {
  prompt user to enter operation code;
  read code;
  switch (code) {
    case 'i': perform insert operation; break;
    case 's': perform search operation; break;
    case 'u': perform update operation; break;
    case 'p': perform print operation; break;
```

```
    case 'q': terminate program;
    default:  print error message;
  }
}
```

It will be convenient to have separate functions perform the insert, search, update, and print operations. Since these functions will all need access to `inventory` and `num_parts`, we might want to make these variables external. As an alternative, we could declare the variables inside `main`, and then pass them to the functions as arguments. From a design standpoint, it's usually better to make variables local to a function rather than making them external (see Section 10.2 if you've forgotten why). In this program, however, putting `inventory` and `num_parts` inside `main` would merely complicate matters.

For reasons that I'll explain later, I've decided to split the program into three files: `inventory.c`, which contains the bulk of the program; `readline.h`, which contains the prototype for the `read_line` function; and `readline.c`, which contains the definition of `read_line`. We'll discuss the latter two files later in this section. For now, let's concentrate on `inventory.c`.

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
  char code;
```

```
  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}

/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 **********************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}

/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **********************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &part_number);
```

```
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}

/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}

/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **********************************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

In the `main` function, the format string `" %c"` allows `scanf` to skip over white space before reading the operation code. The space in the format string is crucial; without it, `scanf` would sometimes read the new-line character that terminated a previous line of input.

The program contains one function, `find_part`, that isn't called from `main`. This "helper" function helps us avoid redundant code and simplify the more important functions. By calling `find_part`, the `insert`, `search`, and `update` functions can locate a part in the database (or simply determine if the part exists).

There's just one detail left: the `read_line` function, which the program uses to read the part name. Section 13.3 discussed the issues that are involved in writing such a function. Unfortunately, the version of `read_line` in that section won't work properly in the current program. Consider what happens when the user inserts a part:

```
Enter part number: 528
Enter part name: Disk drive
```

The user presses the Enter key after entering the part number and again after entering the part name, each time leaving an invisible new-line character that the program must read. For the sake of discussion, let's pretend that these characters are visible:

```
Enter part number: 528¤
Enter part name: Disk drive¤
```

When we call `scanf` to read the part number, it consumes the `5`, `2`, and `8`, but leaves the ¤ character unread. If we try to read the part name using our original `read_line` function, it will encounter the ¤ character immediately and stop reading. This problem is common when numerical input is followed by character input. Our solution will be to write a version of `read_line` that skips white-

space characters before it begins storing characters. Not only will this solve the new-line problem, but it also allows us to avoid storing any blanks that precede the part name.

Since `read_line` is unrelated to the other functions in `inventory.c`, and since it's potentially reusable in other programs, I've decided to separate it from `inventory.c`. The prototype for `read_line` will go in the `readline.h` header file:

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/**********************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 **********************************************************/
int read_line(char str[], int n);

#endif
```

We'll put the definition of `read_line` in the `readline.c` file:

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```

The expression

```
isspace(ch = getchar())
```

controls the first `while` statement. This expression calls `getchar` to read a character, stores the character into `ch`, and then uses the `isspace` function to test whether `ch` is a white-space character. If not, the loop terminates with `ch` containing a character that's not white space. Section 15.3 explains why `ch` has type `int` instead of `char` and why it's good to test for `EOF`.

`isspace` function ➤23.5

16.4 Unions

A *union*, like a structure, consists of one or more members, possibly of different types. However, the compiler allocates only enough space for the largest of the members, which overlay each other within this space. As a result, assigning a new value to one member alters the values of the other members as well.

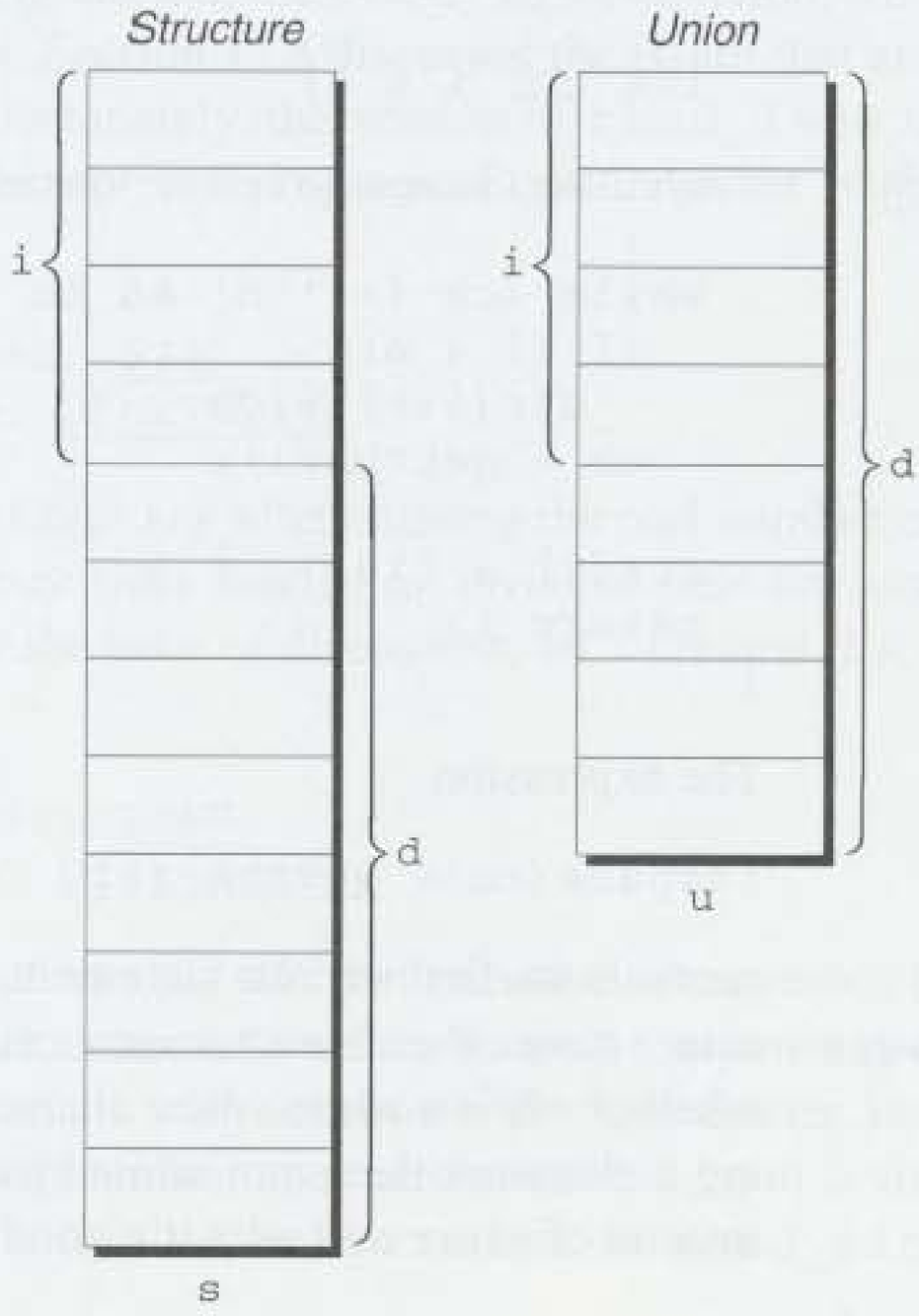
To illustrate the basic properties of unions, let's declare a union variable, `u`, with two members:

```
union {
  int i;
  double d;
} u;
```

Notice how the declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```

In fact, the structure `s` and the union `u` differ in just one way: the members of `s` are stored at *different* addresses in memory, while the members of `u` are stored at the *same* address. Here's what `s` and `u` will look like in memory (assuming that `int` values require four bytes and `double` values take eight bytes):



In the `s` structure, `i` and `d` occupy different memory locations; the total size of `s` is 12 bytes. In the `u` union, `i` and `d` overlap (`i` is really the first four bytes of `d`), so `u` occupies only eight bytes. Also, `i` and `d` have the same address.

Members of a union are accessed in the same way as members of a structure. To store the number 82 in the `i` member of `u`, we would write

```
u.i = 82;
```

To store the value 74.8 in the `d` member, we would write

```
u.d = 74.8;
```

Since the compiler overlays storage for the members of a union, changing one member alters any value previously stored in any of the other members. Thus, if we store a value in `u.d`, any value previously stored in `u.i` will be lost. (If we examine the value of `u.i`, it will appear to be meaningless.) Similarly, changing `u.i` corrupts `u.d`. Because of this property, we can think of `u` as a place to store either `i` *or* `d`, not both. (The structure `s` allows us to store `i` *and* `d`.)

The properties of unions are almost identical to the properties of structures. We can declare union tags and union types in the same way we declare structure tags and types. Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.

Unions can even be initialized in a manner similar to structures. However, only the first member of a union can be given an initial value. For example, we can initialize the `i` member of `u` to 0 in the following way:

```
union {
  int i;
  double d;
} u = {0};
```

Notice the presence of the braces, which are required. The expression inside the braces must be constant. (The rules are slightly different in C99, as we'll see in Section 18.5.)

C99 Designated initializers, a C99 feature that we've previously discussed in the context of arrays and structures, can also be used with unions. A designated initializer allows us to specify which member of a union should be initialized. For example, we can initialize the `d` member of `u` as follows:

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

Only one member can be initialized, but it doesn't have to be the first one.

There are several applications for unions. We'll discuss two of these now. Another application—viewing storage in different ways—is highly machine-dependent, so I'll postpone it until Section 20.3.

Using Unions to Save Space

We'll often use unions as a way to save space in structures. Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog. The catalog carries only three kinds of merchandise: books, mugs, and shirts. Each item has a stock number and a price, as well as other information that depends on the type of the item:

Books: Title, author, number of pages

Mugs: Design

Shirts: Design, colors available, sizes available

Our first design attempt might result in the following structure:

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`. The `colors` and `sizes` members would store encoded combinations of colors and sizes.

Although this structure is perfectly usable, it wastes space, since only part of the information in the structure is common to all items in the catalog. If an item is a book, for example, there's no need to store `design`, `colors`, and `sizes`. By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure. The members of the union will be structures, each containing the data that's needed for a particular kind of catalog item:

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
```

```
        struct {
          char design[DESIGN_LEN+1];
          int colors;
          int sizes;
        } shirt;
    } item;
};
```

Notice that the union (named `item`) is a member of the `catalog_item` structure, and the `book`, `mug`, and `shirt` structures are members of `item`. If `c` is a `catalog_item` structure that represents a book, we can print the book's title in the following way:

```
printf("%s", c.item.book.title);
```

As this example shows, accessing a union that's nested inside a structure can be awkward: to locate a book title, we had to specify the name of a structure (`c`), the name of the union member of the structure (`item`), the name of a structure member of the union (`book`), and then the name of a member of that structure (`title`).

We can use the `catalog_item` structure to illustrate an interesting aspect of unions. Normally, it's not a good idea to store a value into one member of a union and then access the data through a different member, because assigning to one member of a union causes the values of the other members to be undefined. However, the C standard mentions a special case: two or more of the members of the union are structures, and the structures begin with one or more matching members. (These members need to be in the same order and have compatible types, but need not have the same name.) If one of the structures is currently valid, then the matching members in the other structures will also be valid.

Consider the union embedded in the `catalog_item` structure. It contains three structures as members, two of which (`mug` and `shirt`) begin with a matching member (`design`). Now, suppose that we assign a value to one of the `design` members:

```
strcpy(c.item.mug.design, "Cats");
```

The `design` member in the other structure will be defined and have the same value:

```
printf("%s", c.item.shirt.design);   /* prints "Cats" */
```

Using Unions to Build Mixed Data Structures

Unions have another important application: creating data structures that contain a mixture of data of different types. Let's say that we need an array whose elements are a mixture of `int` and `double` values. Since the elements of an array must be of the same type, it seems impossible to create such an array. Using unions, though, it's relatively easy. First, we define a union type whose members represent the different kinds of data to be stored in the array:

```
typedef union {
  int i;
  double d;
} Number;
```

Next, we create an array whose elements are `Number` values:

```
Number number_array[1000];
```

Each element of `number_array` is a `Number` union. A `Number` union can store either an `int` value or a `double` value, making it possible to store a mixture of `int` and `double` values in `number_array`. For example, suppose that we want element 0 of `number_array` to store 5, while element 1 stores 8.395. The following assignments will have the desired effect:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a "Tag Field" to a Union

Unions suffer from a major problem: there's no easy way to tell which member of a union was last changed and therefore contains a meaningful value. Consider the problem of writing a function that displays the value currently stored in a `Number` union. This function might have the following outline:

```
void print_number(Number n)
{
  if (n contains an integer)
    printf("%d", n.i);
  else
    printf("%g", n.d);
}
```

Unfortunately, there's no way for `print_number` to determine whether `n` contains an integer or a floating-point number.

In order to keep track of this information, we can embed the union within a structure that has one other member: a "tag field" or "discriminant," whose purpose is to remind us what's currently stored in the union. In the `catalog_item` structure discussed earlier in this section, `item_type` served this purpose.

Let's convert the `Number` type into a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  int kind;   /* tag field */
  union {
    int i;
    double d;
  } u;
} Number;
```

`Number` has two members, `kind` and `u`. The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified. For example, if `n` is a `Number` variable, an assignment to the `i` member of `u` would have the following appearance:

```
n.kind = INT_KIND;
n.u.i = 82;
```

Notice that assigning to `i` requires that we first select the `u` member of `n`, then the `i` member of `u`.

When we need to retrieve the number stored in a `Number` variable, `kind` will tell us which member of the union was the last to be assigned a value. The `print_number` function can take advantage of this capability:

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```


It's the program's responsibility to change the tag field each time an assignment is made to a member of the union.

16.5 Enumerations

In many programs, we'll need variables that have only a small set of meaningful values. A Boolean variable, for example, should have only two possible values: "true" and "false." A variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades." The obvious way to deal with such a variable is to declare it as an integer and have a set of codes that represent the possible values of the variable:

```
int s;    /* s will store a suit */
...
s = 2;    /* 2 represents "hearts" */
```

Although this technique works, it leaves much to be desired. Someone reading the program can't tell that `s` has only four possible values, and the significance of 2 isn't immediately apparent.

Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
```

Our previous example now becomes easier to read:

```
SUIT s;
...
s = HEARTS;
```

This technique is an improvement, but it's still not the best solution. There's no indication to someone reading the program that the macros represent values of the same "type." If the number of possible values is more than a few, defining a separate macro for each will be tedious. Moreover, the names we've defined—`CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES`—will be removed by the preprocessor, so they won't be available during debugging.

C provides a special kind of type designed specifically for variables that have a small number of possible values. An ***enumerated type*** is a type whose values are listed ("enumerated") by the programmer, who must create a name (an ***enumeration constant***) for each of the values. The following example enumerates the values (`CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES`) that can be assigned to the variables `s1` and `s2`:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

Although enumerations have little in common with structures and unions, they're declared in a similar way. Unlike the members of a structure or union, however, the names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumeration constants are similar to constants created with the `#define` directive, but they're not equivalent. For one thing, enumeration constants are subject to C's scope rules: if an enumeration is declared inside a function, its constants won't be visible outside the function.

Enumeration Tags and Type Names

We'll often need to create names for enumerations, for the same reasons that we name structures and unions. As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.

Enumeration tags resemble structure and union tags. To define the tag `suit`, for example, we could write

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

`suit` variables would be declared in the following way:

```
enum suit s1, s2;
```

As an alternative, we could use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:

```
typedef enum {FALSE, TRUE} Bool;
```

C99 has a built-in Boolean type, of course, so there's no need for a C99 programmer to define a `Bool` type in this way.

Enumerations as Integers

Behind the scenes, C treats enumeration variables and constants as integers. By default, the compiler assigns the integers 0, 1, 2, ... to the constants in a particular enumeration. In our `suit` enumeration, for example, `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` represent 0, 1, 2, and 3, respectively.

We're free to choose different values for enumeration constants if we like. Let's say that we want `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` to stand for 1, 2, 3, and 4. We can specify these numbers when declaring the enumeration:

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
```

The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

It's even legal for two or more enumeration constants to have the same value.

When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant. (The first enumeration constant has the value 0 by default.) In the following enumeration, `BLACK` has the value 0, `LT_GRAY` is 7, `DK_GRAY` is 8, and `WHITE` is 15:

```
enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY, WHITE = 15};
```

Since enumeration values are nothing but thinly disguised integers, C allows us to mix them with ordinary integers:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;    /* i is now 1              */
s = 0;           /* s is now 0 (CLUBS)      */
s++;             /* s is now 1 (DIAMONDS)   */
i = s + 2;       /* i is now 3              */
```

The compiler treats `s` as a variable of some integer type; `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` are just names for the integers 0, 1, 2, and 3.

Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value. For example, we might accidentally store the number 4—which doesn't correspond to any suit—into `s`.

Using Enumerations to Declare "Tag Fields"

Enumerations are perfect for solving a problem that we encountered in Section 16.4: determining which member of a union was the last to be assigned a value. In the `Number` structure, for example, we can make the `kind` member an enumeration instead of an `int`:

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```

The new structure is used in exactly the same way as the old one. The advantages are that we've done away with the `INT_KIND` and `DOUBLE_KIND` macros (they're now enumeration constants), and we've clarified the meaning of `kind`—it's now obvious that `kind` has only two possible values: `INT_KIND` and `DOUBLE_KIND`.

Q & A

Q: **When I tried using the `sizeof` operator to determine the number of bytes in a structure, I got a number that was larger than the sizes of the members added together. How can this be?**

A: Let's look at an example:

```
struct {
  char a;
  int b;
} s;
```

If `char` values occupy one byte and `int` values occupy four bytes, how large is `s`? The obvious answer—five bytes—may not be the correct one. Some computers require that the address of certain data items be a multiple of some number of bytes (typically two, four, or eight, depending on the item's type). To satisfy this requirement, a compiler will "align" the members of a structure by leaving "holes" (unused bytes) between adjacent members. If we assume that data items must

begin on a multiple of four bytes, the `a` member of the `s` structure will be followed by a three-byte hole. As a result, `sizeof(s)` will be 8.

By the way, a structure can have a hole at the end, as well as holes between members. For example, the structure

```
struct {
  int a;
  char b;
} s;
```

might have a three-byte hole after the `b` member.

Q: Can there be a "hole" at the beginning of a structure?

A: No. The C standard specifies that holes are allowed only *between* members or *after* the last member. One consequence is that a pointer to the first member of a structure is guaranteed to be the same as a pointer to the entire structure. (Note, however, that the two pointers won't have the same type.)

Q: Why isn't it legal to use the == operator to test whether two structures are equal? [p. 382]

A: This operation was left out of C because there's no way to implement it that would be consistent with the language's philosophy. Comparing structure members one by one would be too inefficient. Comparing all bytes in the structures would be better (many computers have special instructions that can perform such a comparison rapidly). If the structures contain holes, however, comparing bytes could yield an incorrect answer; even if corresponding members have identical values, leftover data stored in the holes might be different. The problem could be solved by having the compiler ensure that holes always contain the same value (zero, say). Initializing holes would impose a performance penalty on all programs that use structures, however, so it's not feasible.

Q: Why does C provide two ways to name structure types (tags and `typedef` names)? [p. 382]

A: C originally lacked `typedef`, so tags were the only technique available for naming structure types. When `typedef` was added, it was too late to remove tags. Besides, a tag is still necessary when a member of a structure points to a structure of the same type (see the `node` structure of Section 17.5).

Q: Can a structure have both a tag *and* a `typedef` name? [p. 384]

A: Yes. In fact, the tag and the `typedef` name can even be the same, although that's not required:

```
typedef struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part;
```

Q: How can I share a structure type among several files in a program?

A: Put a declaration of the structure tag (or a `typedef`, if you prefer) in a header file, then include the header file where the structure is needed. To share the `part` structure, for example, we'd put the following lines in a header file:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

Notice that we're declaring only the structure *tag*, not variables of this type.

Incidentally, a header file that contains a declaration of a structure tag or structure type may need protection against multiple inclusion. Declaring a tag or `typedef` name twice in the same file is an error. Similar remarks apply to unions and enumerations.

protecting header files ➤ 15.2

Q: If I include the declaration of the `part` structure into two different files, will `part` variables in one file be of the same type as `part` variables in the other file?

A: Technically, no. However, the C standard says that the `part` variables in one file have a type that's compatible with the type of the `part` variables in the other file. Variables with compatible types can be assigned to each other, so there's little practical difference between types being "compatible" and being "the same."

C99 The rules for structure compatibility in C89 and C99 are slightly different. In C89, structures defined in different files are compatible if their members have the same names and appear in the same order, with corresponding members having compatible types. C99 goes one step further: it requires that either both structures have the same tag or neither has a tag.

Similar compatibility rules apply to unions and enumerations (with the same difference between C89 and C99).

Q: Is it legal to have a pointer to a compound literal?

A: Yes. Consider the `print_part` function of Section 16.2. Currently, the parameter to this function is a `part` structure. The function would be more efficient if it were modified to accept a *pointer* to a `part` structure instead. Using the function to print a compound literal would then be done by prefixing the argument with the & (address) operator:

```
print_part(&(struct part) {528, "Disk drive", 10});
```

Q: Allowing a pointer to a compound literal would seem to make it possible to modify the literal. Is that the case?

C99

A: Yes. Compound literals are lvalues that can be modified, although doing so is rare.

Q: I saw a program in which the last constant in an enumeration was followed by a comma, like this:

```
enum gray_values {
  BLACK = 0,
  DARK_GRAY = 64,
  GRAY = 128,
  LIGHT_GRAY = 192,
};
```

Is this practice legal?

A: This practice is indeed legal in C99 (and is supported by some pre-C99 compilers as well). Allowing a "trailing comma" makes enumerations easier to modify, because we can add a constant to the end of an enumeration without changing existing lines of code. For example, we might want to add `WHITE` to our enumeration:

C99

```
enum gray_values {
  BLACK = 0,
  DARK_GRAY = 64,
  GRAY = 128,
  LIGHT_GRAY = 192,
  WHITE = 255,
};
```

The comma after the definition of `LIGHT_GRAY` makes it easy to add `WHITE` to the end of the list.

One reason for this change is that C89 allows trailing commas in initializers, so it seemed inconsistent not to allow the same flexibility in enumerations. Incidentally, C99 also allows trailing commas in compound literals.

C99

Q: Can the values of an enumerated type be used as subscripts?

A: Yes, indeed. They are integers and have—by default—values that start at 0 and count upward, so they make great subscripts. In C99, moreover, enumeration constants can be used as subscripts in designated initializers. Here's an example:

C99

```
enum weekdays {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
const char *daily_specials[] = {
  [MONDAY] = "Beef ravioli",
  [TUESDAY] = "BLTs",
  [WEDNESDAY] = "Pizza",
  [THURSDAY] = "Chicken fajitas",
  [FRIDAY] = "Macaroni and cheese"
};
```

Exercises

Section 16.1

1. In the following declarations, the `x` and `y` structures have members named `x` and `y`:

```
struct { int x, y; } x;
struct { int x, y; } y;
```

Are these declarations legal on an individual basis? Could both declarations appear as shown in a program? Justify your answer.

W 2. (a) Declare structure variables named `c1`, `c2`, and `c3`, each having members `real` and `imaginary` of type `double`.

(b) Modify the declaration in part (a) so that `c1`'s members initially have the values 0.0 and 1.0, while `c2`'s members are 1.0 and 0.0 initially. (`c3` is not initialized.)

(c) Write statements that copy the members of `c2` into `c1`. Can this be done in one statement, or does it require two?

(d) Write statements that add the corresponding members of `c1` and `c2`, storing the result in `c3`.

Section 16.2

3. (a) Show how to declare a tag named `complex` for a structure with two members, `real` and `imaginary`, of type `double`.

(b) Use the `complex` tag to declare variables named `c1`, `c2`, and `c3`.

(c) Write a function named `make_complex` that stores its two arguments (both of type `double`) in a `complex` structure, then returns the structure.

(d) Write a function named `add_complex` that adds the corresponding members of its arguments (both `complex` structures), then returns the result (another `complex` structure).

W 4. Repeat Exercise 3, but this time using a *type* named `Complex`.

5. Write the following functions, assuming that the `date` structure contains three members: `month`, `day`, and `year` (all of type `int`).

(a) `int day_of_year(struct date d);`

Returns the day of the year (an integer between 1 and 366) that corresponds to the date `d`.

(b) `int compare_dates(struct date d1, struct date d2);`

Returns –1 if `d1` is an earlier date than `d2`, +1 if `d1` is a later date than `d2`, and 0 if `d1` and `d2` are the same.

6. Write the following function, assuming that the `time` structure contains three members: `hours`, `minutes`, and `seconds` (all of type `int`).

`struct time split_time(long total_seconds);`

`total_seconds` is a time represented as the number of seconds since midnight. The function returns a structure containing the equivalent time in hours (0–23), minutes (0–59), and seconds (0–59).

7. Assume that the `fraction` structure contains two members: `numerator` and `denominator` (both of type `int`). Write functions that perform the following operations on fractions:

(a) Reduce the fraction `f` to lowest terms. *Hint:* To reduce a fraction to lowest terms, first compute the greatest common divisor (GCD) of the numerator and denominator. Then divide both the numerator and denominator by the GCD.
(b) Add the fractions `f1` and `f2`.
(c) Subtract the fraction `f2` from the fraction `f1`.
(d) Multiply the fractions `f1` and `f2`.
(e) Divide the fraction `f1` by the fraction `f2`.

The fractions `f`, `f1`, and `f2` will be arguments of type `struct fraction`; each function will return a value of type `struct fraction`. The fractions returned by the functions in parts (b)–(e) should be reduced to lowest terms. *Hint:* You may use the function from part (a) to help write the functions in parts (b)–(e).

8. Let `color` be the following structure:

```
struct color {
  int red;
  int green;
  int blue;
};
```

(a) Write a declaration for a `const` variable named `MAGENTA` of type `struct color` whose members have the values 255, 0, and 255, respectively.

(b) (C99) Repeat part (a), but use a designated initializer that doesn't specify the value of `green`, allowing it to default to 0.

9. Write the following functions. (The `color` structure is defined in Exercise 8.)

(a) `struct color make_color(int red, int green, int blue);`

Returns a `color` structure containing the specified red, green, and blue values. If any argument is less than zero, the corresponding member of the structure will contain zero instead. If any argument is greater than 255, the corresponding member of the structure will contain 255.

(b) `int getRed(struct color c);`

Returns the value of `c`'s `red` member.

(c) `bool equal_color(struct color color1, struct color color2);`

Returns `true` if the corresponding members of `color1` and `color2` are equal.

(d) `struct color brighter(struct color c);`

Returns a `color` structure that represents a brighter version of the color `c`. The structure is identical to `c`, except that each member has been divided by 0.7 (with the result truncated to an integer). However, there are three special cases: (1) If all members of `c` are zero, the function returns a color whose members all have the value 3. (2) If any member of `c` is greater than 0 but less than 3, it is replaced by 3 before the division by 0.7. (3) If dividing by 0.7 causes a member to exceed 255, it is reduced to 255.

(e) `struct color darker(struct color c);`

Returns a `color` structure that represents a darker version of the color `c`. The structure is identical to `c`, except that each member has been multiplied by 0.7 (with the result truncated to an integer).

Section 16.3

10. The following structures are designed to store information about objects on a graphics screen:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
```

A `point` structure stores the *x* and *y* coordinates of a point on the screen. A `rectangle` structure stores the coordinates of the upper left and lower right corners of a rectangle. Write functions that perform the following operations on a `rectangle` structure `r` passed as an argument:

(a) Compute the area of `r`.
(b) Compute the center of `r`, returning it as a `point` value. If either the *x* or *y* coordinate of the center isn't an integer, store its truncated value in the `point` structure.
(c) Move `r` by `x` units in the *x* direction and `y` units in the *y* direction, returning the modified version of `r`. (`x` and `y` are additional arguments to the function.)
(d) Determine whether a point `p` lies within `r`, returning `true` or `false`. (`p` is an additional argument of type `struct point`.)

Section 16.4

W 11. Suppose that `s` is the following structure:

```
struct {
  double a;
  union {
    char b[4];
    double c;
    int d;
  } e;
  char f[4];
} s;
```

If `char` values occupy one byte, `int` values occupy four bytes, and `double` values occupy eight bytes, how much space will a C compiler allocate for `s`? (Assume that the compiler leaves no "holes" between members.)

12. Suppose that `u` is the following union:

```
union {
  double a;
  struct {
    char b[4];
    double c;
    int d;
  } e;
  char f[4];
} u;
```

If `char` values occupy one byte, `int` values occupy four bytes, and `double` values occupy eight bytes, how much space will a C compiler allocate for `u`? (Assume that the compiler leaves no "holes" between members.)

13. Suppose that `s` is the following structure (`point` is a structure tag declared in Exercise 10):

```
struct shape {
  int shape_kind;          /* RECTANGLE or CIRCLE */
  struct point center;     /* coordinates of center */
  union {
    struct {
      int height, width;
    } rectangle;
    struct {
      int radius;
    } circle;
  } u;
} s;
```

If the value of `shape_kind` is `RECTANGLE`, the `height` and `width` members store the dimensions of a rectangle. If the value of `shape_kind` is `CIRCLE`, the `radius` member stores the radius of a circle. Indicate which of the following statements are legal, and show how to repair the ones that aren't:

(a) `s.shape_kind = RECTANGLE;`
(b) `s.center.x = 10;`
(c) `s.height = 25;`
(d) `s.u.rectangle.width = 8;`
(e) `s.u.circle = 5;`
(f) `s.u.radius = 5;`

W 14. Let `shape` be the structure tag declared in Exercise 13. Write functions that perform the following operations on a `shape` structure `s` passed as an argument:

(a) Compute the area of `s`.

(b) Move `s` by `x` units in the *x* direction and `y` units in the *y* direction, returning the modified version of `s`. (`x` and `y` are additional arguments to the function.)

(c) Scale `s` by a factor of `c` (a `double` value), returning the modified version of `s`. (`c` is an additional argument to the function.)

Section 16.5

W 15. (a) Declare a tag for an enumeration whose values represent the seven days of the week.

(b) Use `typedef` to define a name for the enumeration of part (a).

16. Which of the following statements about enumeration constants are true?

(a) An enumeration constant may represent any integer specified by the programmer.

(b) Enumeration constants have exactly the same properties as constants created using `#define`.

(c) Enumeration constants have the values 0, 1, 2, … by default.

(d) All constants in an enumeration must have different values.

(e) Enumeration constants may be used as integers in expressions.

W 17. Suppose that `b` and `i` are declared as follows:

```
enum {FALSE, TRUE} b;
int i;
```

Which of the following statements are legal? Which ones are "safe" (always yield a meaningful result)?

```
(a) b = FALSE;
(b) b = i;
(c) b++;
(d) i = b;
(e) i = 2 * b + 1;
```

18. (a) Each square of a chessboard can hold one piece—a pawn, knight, bishop, rook, queen, or king—or it may be empty. Each piece is either black or white. Define two enumerated types: `Piece`, which has seven possible values (one of which is "empty"), and `Color`, which has two.

(b) Using the types from part (a), define a structure type named `Square` that can store both the type of a piece and its color.

(c) Using the `Square` type from part (b), declare an 8×8 array named `board` that can store the entire contents of a chessboard.

(d) Add an initializer to the declaration in part (c) so that `board`'s initial value corresponds to the usual arrangement of pieces at the start of a chess game. A square that's not occupied by a piece should have an "empty" piece value and the color black.

19. Declare a structure with the following members whose tag is `pinball_machine`:

`name` – a string of up to 40 characters
`year` – an integer (representing the year of manufacture)
`type` – an enumeration with the values `EM` (electromechanical) and `SS` (solid state)
`players` – an integer (representing the maximum number of players)

20. Suppose that the `direction` variable is declared in the following way:

```
enum {NORTH, SOUTH, EAST, WEST} direction;
```

Let `x` and `y` be `int` variables. Write a `switch` statement that tests the value of `direction`, incrementing `x` if `direction` is `EAST`, decrementing `x` if `direction` is `WEST`, incrementing `y` if `direction` is `SOUTH`, and decrementing `y` if `direction` is `NORTH`.

21. What are the integer values of the enumeration constants in each of the following declarations?

(a) `enum {NUL, SOH, STX, ETX};`
(b) `enum {VT = 11, FF, CR};`
(c) `enum {SO = 14, SI, DLE, CAN = 24, EM};`
(d) `enum {ENQ = 45, ACK, BEL, LF = 37, ETB, ESC};`

22. Let `chess_pieces` be the following enumeration:

```
enum chess_pieces {KING, QUEEN, ROOK, BISHOP, KNIGHT, PAWN};
```

(a) Write a declaration (including an initializer) for a constant array of integers named `piece_value` that stores the numbers 200, 9, 5, 3, 3, and 1, representing the value of each chess piece, from king to pawn. (The king's value is actually infinite, since "capturing" the king (checkmate) ends the game, but some chess-playing software assigns the king a large value such as 200.)

(b) (C99) Repeat part (a), but use a designated initializer to initialize the array. Use the enumeration constants in `chess_pieces` as subscripts in the designators. (*Hint:* See the last question in Q&A for an example.)

Programming Projects

1. Write a program that asks the user to enter an international dialing code and then looks it up in the `country_codes` array (see Section 16.3). If it finds the code, the program should display the name of the corresponding country; if not, the program should print an error message.

2. Modify the `inventory.c` program of Section 16.3 so that the `p` (print) operation displays the parts sorted by part number.

3. Modify the `inventory.c` program of Section 16.3 by making `inventory` and `num_parts` local to the `main` function.

4. Modify the `inventory.c` program of Section 16.3 by adding a `price` member to the `part` structure. The `insert` function should ask the user for the price of a new item. The `search` and `print` functions should display the price. Add a new command that allows the user to change the price of a part.

5. Modify Programming Project 8 from Chapter 5 so that the times are stored in a single array. The elements of the array will be structures, each containing a departure time and the corresponding arrival time. (Each time will be an integer, representing the number of minutes since midnight.) The program will use a loop to search the array for the departure time closest to the time entered by the user.

6. Modify Programming Project 9 from Chapter 5 so that each date entered by the user is stored in a `date` structure (see Exercise 5). Incorporate the `compare_dates` function of Exercise 5 into your program.

17 Advanced Uses of Pointers

One can only display complex information in the mind. Like seeing, movement or flow or alteration of view is more important than the static picture, no matter how lovely.

In previous chapters, we've seen two important uses of pointers. Chapter 11 showed how using a pointer to a variable as a function argument allows the function to modify the variable. Chapter 12 showed how to process arrays by performing arithmetic on pointers to array elements. This chapter completes our coverage of pointers by examining two additional applications: dynamic storage allocation and pointers to functions.

Using dynamic storage allocation, a program can obtain blocks of memory as needed during execution. Section 17.1 explains the basics of dynamic storage allocation. Section 17.2 discusses dynamically allocated strings, which provide more flexibility than ordinary character arrays. Section 17.3 covers dynamic storage allocation for arrays in general. Section 17.4 deals with the issue of storage deallocation—releasing blocks of dynamically allocated memory when they're no longer needed.

Dynamically allocated structures play a big role in C programming, since they can be linked together to form lists, trees, and other highly flexible data structures. Section 17.5 focuses on linked lists, the most fundamental linked data structure. One of the issues that arises in this section—the concept of a "pointer to a pointer"—is important enough to warrant a section of its own (Section 17.6).

Section 17.7 introduces pointers to functions, a surprisingly useful concept. Some of C's most powerful library functions expect function pointers as arguments. We'll examine one of these functions, `qsort`, which is capable of sorting any array.

The last two sections discuss pointer-related features that first appeared in C99: restricted pointers (Section 17.8) and flexible array members (Section 17.9). These features are primarily of interest to advanced C programmers, so both sections can be safely be skipped by the beginner.

17.1 Dynamic Storage Allocation

variable-length arrays ➤8.3

C's data structures are normally fixed in size. For example, the number of elements in an array is fixed once the program has been compiled. (In C99, the length of a variable-length array is determined at run time, but it remains fixed for the rest of the array's lifetime.) Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program; we can't change the sizes without modifying the program and compiling it again.

Consider the `inventory` program of Section 16.3, which allows the user to add parts to a database. The database is stored in an array of length 100. To enlarge the capacity of the database, we can increase the size of the array and recompile the program. But no matter how large we make the array, there's always the possibility that it will fill up. Fortunately, all is not lost. C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution. Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Although it's available for all types of data, dynamic storage allocation is used most often for strings, arrays, and structures. Dynamically allocated structures are of particular interest, since we can link them together to form lists, trees, and other data structures.

Memory Allocation Functions

<stdlib.h> header ➤26.2

To allocate storage dynamically, we'll need to call one of the three memory allocation functions declared in the `<stdlib.h>` header:

- `malloc`—Allocates a block of memory but doesn't initialize it.
- `calloc`—Allocates a block of memory and clears it.
- `realloc`—Resizes a previously allocated block of memory.

Of the three, `malloc` is the most used. It's more efficient than `calloc`, since it doesn't have to clear the memory block that it allocates.

When we call a memory allocation function to request a block of memory, the function has no idea what type of data we're planning to store in the block, so it can't return a pointer to an ordinary type such as `int` or `char`. Instead, the function returns a value of type `void *`. A `void *` value is a "generic" pointer—essentially, just a memory address.

Null Pointers

When a memory allocation function is called, there's always a possibility that it won't be able to locate a block of memory large enough to satisfy our request. If

that should happen, the function will return a ***null pointer.*** A null pointer is a "pointer to nothing"—a special value that can be distinguished from all valid pointers. After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

It's the programmer's responsibility to test the return value of any memory allocation function and take appropriate action if it's a null pointer. The effect of attempting to access memory through a null pointer is undefined; the program may crash or behave unpredictably.

Q&A The null pointer is represented by a macro named `NULL`, so we can test `malloc`'s return value in the following way:

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

Some programmers combine the call of `malloc` with the `NULL` test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```

C99 The `NULL` macro is defined in six headers: `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`. (The C99 header `<wchar.h>` also defines `NULL`.) As long as one of these headers is included in a program, the compiler will recognize `NULL`. A program that uses any of the memory allocation functions will include `<stdlib.h>`, of course, making `NULL` available.

In C, pointers test true or false in the same way as numbers. All non-null pointers test true; only null pointers are false. Thus, instead of writing

```
if (p == NULL) ...
```

we could write

```
if (!p) ...
```

and instead of writing

```
if (p != NULL) ...
```

we could write

```
if (p) ...
```

As a matter of style, I prefer the explicit comparison with `NULL`.

17.2 Dynamically Allocated Strings

Dynamic storage allocation is often useful for working with strings. Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be. By allocating strings dynamically, we can postpone the decision until the program is running.

Using `malloc` to Allocate Memory for a String

The `malloc` function has the following prototype:

```
void *malloc(size_t size);
```

`malloc` allocates a block of `size` bytes and returns a pointer to it. Note that `size` has type `size_t`, an unsigned integer type defined in the C library. Unless we're allocating a very large block of memory, we can just think of `size` as an ordinary integer.

`size_t` type ➤7.6

Using `malloc` to allocate memory for a string is easy, because C guarantees that a `char` value requires exactly one byte of storage (`sizeof(char)` is 1, in other words). To allocate space for a string of `n` characters, we'd write

```
p = malloc(n + 1);
```

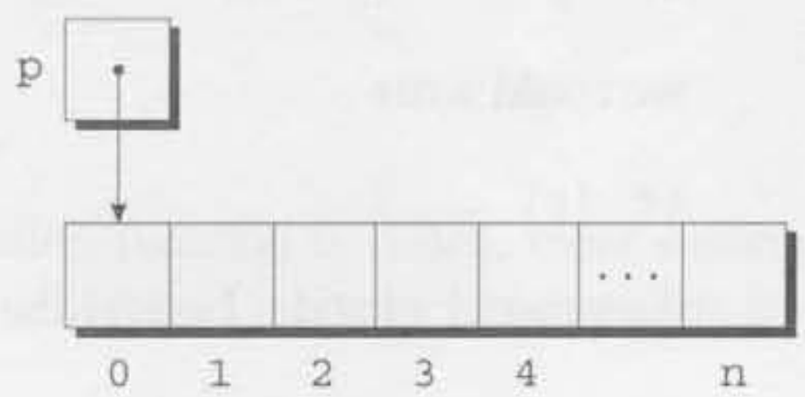
where `p` is a `char *` variable. (The argument is `n + 1` rather than `n` to allow room for the null character.) The generic pointer that `malloc` returns will be converted to `char *` when the assignment is performed; no cast is necessary. (In general, we can assign a `void *` value to a variable of any pointer type and vice versa.) Nevertheless, some programmers prefer to cast `malloc`'s return value:

Q&A

```
p = (char *) malloc(n + 1);
```

When using `malloc` to allocate space for a string, don't forget to include room for the null character.

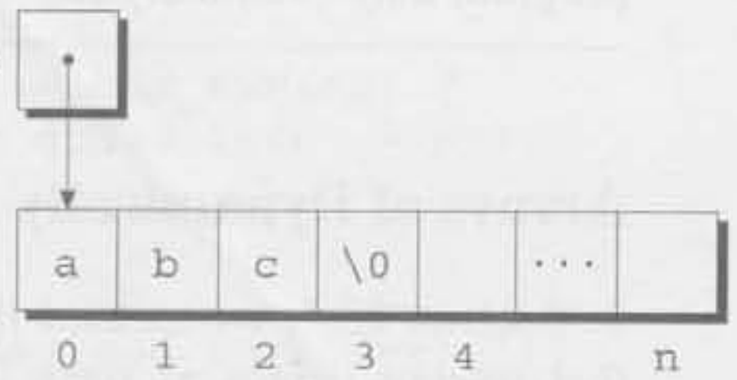
Memory allocated using `malloc` isn't cleared or initialized in any way, so `p` will point to an uninitialized array of `n + 1` characters:



Calling `strcpy` is one way to initialize this array:

```
strcpy(p, "abc");
```

The first four characters in the array will now be `a`, `b`, `c`, and `\0`:



Using Dynamic Storage Allocation in String Functions

Dynamic storage allocation makes it possible to write functions that return a pointer to a "new" string—a string that didn't exist before the function was called. Consider the problem of writing a function that concatenates two strings without changing either one. C's standard library doesn't include such a function (`strcat` isn't quite what we want, since it modifies one of the strings passed to it), but we can easily write our own.

Our function will measure the lengths of the two strings to be concatenated, then call `malloc` to allocate just the right amount of space for the result. The function next copies the first string into the new space and then calls `strcat` to concatenate the second string.

```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

If `malloc` returns a null pointer, `concat` prints an error message and terminates the program. That's not always the right action to take; some programs need to recover from memory allocation failures and continue running.

Here's how the `concat` function might be called:

```
p = concat("abc", "def");
```

After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array. The array is seven characters long, including the null character at the end.

Functions such as `concat` that dynamically allocate storage must be used with care. When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release the space that the string occupies. If we don't, the program may eventually run out of memory.

`free` function ➤ 17.4

Arrays of Dynamically Allocated Strings

In Section 13.7, we tackled the problem of storing strings in an array. We found that storing strings as rows in a two-dimensional array of characters can waste space, so we tried setting up an array of pointers to string literals. The techniques of Section 13.7 work just as well if the elements of an array are pointers to dynamically allocated strings. To illustrate this point, let's rewrite the `remind.c` program of Section 13.5, which prints a one-month list of daily reminders.

PROGRAM

Printing a One-Month Reminder List (Revisited)

The original `remind.c` program stores the reminder strings in a two-dimensional array of characters, with each row of the array containing one string. After the program reads a day and its associated reminder, it searches the array to determine where the day belongs, using `strcmp` to do comparisons. It then uses `strcpy` to move all strings below that point down one position. Finally, the program copies the day into the array and calls `strcat` to append the reminder to the day.

In the new program (`remind2.c`), the array will be one-dimensional; its elements will be pointers to dynamically allocated strings. Switching to dynamically allocated strings in this program will have two primary advantages. First, we can use space more efficiently by allocating the exact number of characters needed to store a reminder, rather than storing the reminder in a fixed number of characters as the original program does. Second, we won't need to call `strcpy` to move existing reminder strings in order to make room for a new reminder. Instead, we'll merely move *pointers* to strings.

Here's the new program, with changes in **bold.** Switching from a two-dimensional array to an array of pointers turns out to be remarkably easy: we'll only need to change eight lines of the program.

remind2.c

```
/* Prints a one-month reminder list (dynamic string version) */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */

int read_line(char str[], int n);
```

```
int main(void)
{
  char *reminders[MAX_REMIND];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }

    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      reminders[j] = reminders[j-1];

    reminders[i] = malloc(2 + strlen(msg_str) + 1);
    if (reminders[i] == NULL) {
      printf("-- No space left --\n");
      break;
    }

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}

int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

17.3 Dynamically Allocated Arrays

Dynamically allocated arrays have the same advantages as dynamically allocated strings (not surprisingly, since strings *are* arrays). When we're writing a program, it's often difficult to estimate the proper size for an array; it would be more convenient to wait until the program is run to decide how large the array should be. C solves this problem by allowing a program to allocate space for an array during execution, then access the array through a pointer to its first element. The close relationship between arrays and pointers, which we explored in Chapter 12, makes a dynamically allocated array just as easy to use as an ordinary array.

Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since it initializes the memory that it allocates. The `realloc` function allows us to make an array "grow" or "shrink" as needed.

Using `malloc` to Allocate Storage for an Array

We can use `malloc` to allocate space for an array in much the same way we used it to allocate space for a string. The primary difference is that the elements of an arbitrary array won't necessarily be one byte long, as they are in a string. As a result, we'll need to use the `sizeof` operator to calculate the amount of space required for each element.

`sizeof` operator ➤7.6

Suppose we're writing a program that needs an array of `n` integers, where `n` is to be computed during the execution of the program. We'll first declare a pointer variable:

```
int *a;
```

Once the value of `n` is known, we'll have the program call `malloc` to allocate space for the array:

```
a = malloc(n * sizeof(int));
```

Always use `sizeof` when calculating how much space is needed for an array. Failing to allocate enough memory can have severe consequences. Consider the following attempt to allocate space for an array of `n` integers:

```
a = malloc(n * 2);
```

If `int` values are larger than two bytes (as they are on most computers), `malloc` won't allocate a large enough block of memory. When we later try to access elements of the array, the program may crash or behave erratically.

Once it points to a dynamically allocated block of memory, we can ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relation-

ship between arrays and pointers in C. For example, we could use the following loop to initialize the array that a points to:

```
for (i = 0; i < n; i++)
  a[i] = 0;
```

We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

The calloc Function

Although the malloc function can be used to allocate memory for an array, C provides an alternative—the calloc function—that's sometimes better. calloc has the following prototype in <stdlib.h>:

```
void *calloc(size_t nmemb, size_t size);
```

calloc allocates space for an array with nmemb elements, each of which is size bytes long; it returns a null pointer if the requested space isn't available. After allocating the memory, calloc initializes it by setting all bits to 0. For example, the following call of calloc allocates space for an array of n integers, which are all guaranteed to be zero initially:

Q&A

```
a = calloc(n, sizeof(int));
```

Since calloc clears the memory that it allocates but malloc doesn't, we may occasionally want to use calloc to allocate space for an object other than an array. By calling calloc with 1 as its first argument, we can allocate space for a data item of any type:

```
struct point { int x, y; } *p;

p = calloc(1, sizeof(struct point));
```

After this statement has been executed, p will point to a structure whose x and y members have been set to zero.

The realloc Function

Once we've allocated memory for an array, we may later find that it's too large or too small. The realloc function can resize the array to better suit our needs. The following prototype for realloc appears in <stdlib.h>:

```
void *realloc(void *ptr, size_t size);
```

When realloc is called, ptr must point to a memory block obtained by a previous call of malloc, calloc, or realloc. The size parameter represents the new size of the block, which may be larger or smaller than the original size. Although realloc doesn't require that ptr point to memory that's being used as an array, in practice it usually does.

Be sure that a pointer passed to `realloc` came from a previous call of `malloc`, `calloc`, or `realloc`. If it didn't, calling `realloc` causes undefined behavior.

The C standard spells out a number of rules concerning the behavior of `realloc`:

- When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
- If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
- If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
- If `realloc` is called with `0` as its second argument, it frees the memory block.

The C standard stops short of specifying exactly how `realloc` works. Still, we expect it to be reasonably efficient. When asked to reduce the size of a memory block, `realloc` should shrink the block "in place," without moving the data stored in the block. By the same token, `realloc` should always attempt to expand a memory block without moving it. If it's unable to enlarge the block (because the bytes following the block are already in use for some other purpose), `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.

Once `realloc` has returned, be sure to update all pointers to the memory block, since it's possible that `realloc` has moved the block elsewhere.

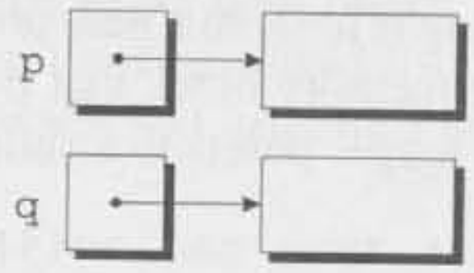
17.4 Deallocating Storage

`malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap***. Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.

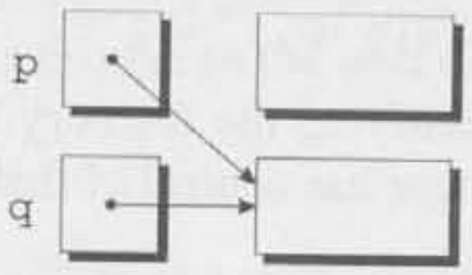
To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space. Consider the following example:

```
p = malloc(...);
q = malloc(...);
p = q;
```

After the first two statements have been executed, `p` points to one memory block, while `q` points to another:



After `q` is assigned to `p`, both variables now point to the second memory block:



There are no pointers to the first block (shaded), so we'll never be able to use it again.

A block of memory that's no longer accessible to a program is said to be ***garbage.*** A program that leaves garbage behind has a ***memory leak.*** Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't. Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The `free` Function

The `free` function has the following prototype in `<stdlib.h>`:

```
void free(void *ptr);
```

Using `free` is easy; we simply pass it a pointer to a memory block that we no longer need:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

Calling `free` releases the block of memory that `p` points to. This block is now available for reuse in subsequent calls of `malloc` or other memory allocation functions.

The argument to `free` must be a pointer that was previously returned by a memory allocation function. (The argument may also be a null pointer, in which case the call of `free` has no effect.) Passing `free` a pointer to any other object (such as a variable or array element) causes undefined behavior.

The "Dangling Pointer" Problem

Although the `free` function allows us to reclaim memory that's no longer needed, using it leads to a new problem: ***dangling pointers.*** The call `free(p)` deallocates the memory block that `p` points to, but doesn't change `p` itself. If we forget that `p` no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
...
free(p);
...
strcpy(p, "abc");    /*** WRONG ***/
```

Modifying the memory that `p` points to is a serious error, since our program no longer has control of that memory.

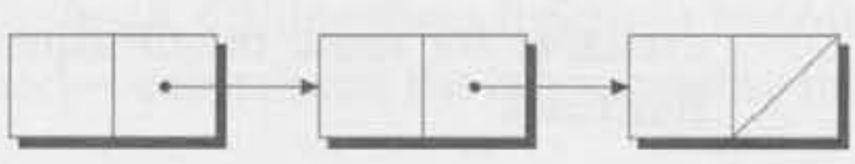
Attempting to access or modify a deallocated memory block causes undefined behavior. Trying to modify a deallocated memory block is likely to have disastrous consequences that may include a program crash.

Dangling pointers can be hard to spot, since several pointers may point to the same block of memory. When the block is freed, all the pointers are left dangling.

17.5 Linked Lists

Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures. We'll look at linked lists in this section; a discussion of other linked data structures is beyond the scope of this book. For more information, consult a book such as Robert Sedgewick's *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching,* Third Edition (Reading, Mass.: Addison-Wesley, 1998).

A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain:



The last node in the list contains a null pointer, shown here as a diagonal line.

In previous chapters, we've used an array whenever we've needed to store a collection of data items; linked lists give us an alternative. A linked list is more flexible than an array: we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed. On the other hand, we lose the "random access" capability of an array. Any element of an array can be accessed in the same

amount of time; accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it's near the end.

This section describes how to set up a linked list in C. It also shows how to perform several common operations on linked lists: inserting a node at the beginning of a list, searching for a node, and deleting a node.

Declaring a Node Type

To set up a linked list, the first thing we'll need is a structure that represents a single node in the list. For simplicity, let's assume that a node contains nothing but an integer (the node's data) plus a pointer to the next node in the list. Here's what our node structure will look like:

```
struct node {
  int value;              /* data stored in the node  */
  struct node *next;      /* pointer to the next node */
};
```

Notice that the `next` member has type `struct node *`, which means that it can store a pointer to a `node` structure. There's nothing special about the name `node`, by the way; it's just an ordinary structure tag.

One aspect of the `node` structure deserves special mention. As Section 16.2 explained, we normally have the option of using either a tag or a `typedef` name to define a name for a particular kind of structure. However, when a structure has a member that points to the same kind of structure, as `node` does, we're required to use a structure tag. Without the `node` tag, we'd have no way to declare the type of `next`.

Q&A

Now that we have the `node` structure declared, we'll need a way to keep track of where the list begins. In other words, we'll need a variable that always points to the first node in the list. Let's name the variable `first`:

```
struct node *first = NULL;
```

Setting `first` to `NULL` indicates that the list is initially empty.

Creating a Node

As we construct a linked list, we'll want to create nodes one by one, adding each to the list. Creating a node requires three steps:

1. Allocate memory for the node.
2. Store data in the node.
3. Insert the node into the list.

We'll concentrate on the first two steps for now.

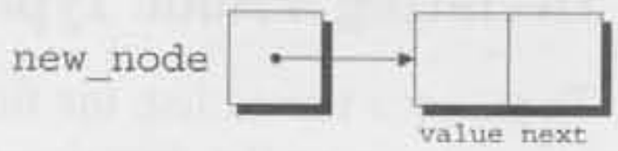
When we create a node, we'll need a variable that can point to the node temporarily, until it's been inserted into the list. Let's call this variable `new_node`:

```
struct node *new_node;
```

We'll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

```
new_node = malloc(sizeof(struct node));
```

`new_node` now points to a block of memory just large enough to hold a `node` structure:



Be careful to give `sizeof` the name of the *type* to be allocated, not the name of a *pointer* to that type:

```
new_node = malloc(sizeof(new_node));    /*** WRONG ***/
```

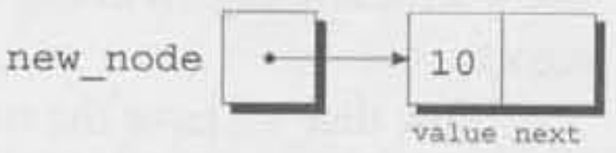
The program will still compile, but `malloc` will allocate only enough memory for a *pointer* to a `node` structure. The likely result is a crash later, when the program attempts to store data in the node that `new_node` is presumably pointing to.

Q&A

Next, we'll store data in the `value` member of the new node:

```
(*new_node).value = 10;
```

Here's how the picture will look after this assignment:



To access the `value` member of the node, we've applied the indirection operator `*` (to reference the structure to which `new_node` points), then the selection operator `.` (to select a member of the structure). The parentheses around `*new_node` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

table of operators ➤ Appendix A

The -> Operator

Before we go on to the next step, inserting a new node into a list, let's take a moment to discuss a useful shortcut. Accessing a member of a structure using a pointer is so common that C provides a special operator just for this purpose. This operator, known as ***right arrow selection,*** is a minus sign followed by >. Using the `->` operator, we can write

```
new_node->value = 10;
```

instead of

```
(*new_node).value = 10;
```

The `->` operator is a combination of the `*` and `.` operators; it performs indirection on `new_node` to locate the structure that it points to, then selects the `value` member of the structure.

lvalues ➤4.2

The `->` operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed. We've just seen an example in which `new_node->value` appears on the left side of an assignment. It could just as easily appear in a call of `scanf`:

```
scanf("%d", &new_node->value);
```

Notice that the `&` operator is still required, even though `new_node` is a pointer. Without the `&`, we'd be passing `scanf` the *value* of `new_node->value`, which has type `int`.

Inserting a Node at the Beginning of a Linked List

One of the advantages of a linked list is that nodes can be added at any point in the list: at the beginning, at the end, or anywhere in the middle. The beginning of a list is the easiest place to insert a node, however, so let's focus on that case.

If `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list, then we'll need two statements to insert the node into the list. First, we'll modify the new node's `next` member to point to the node that was previously at the beginning of the list:

```
new_node->next = first;
```

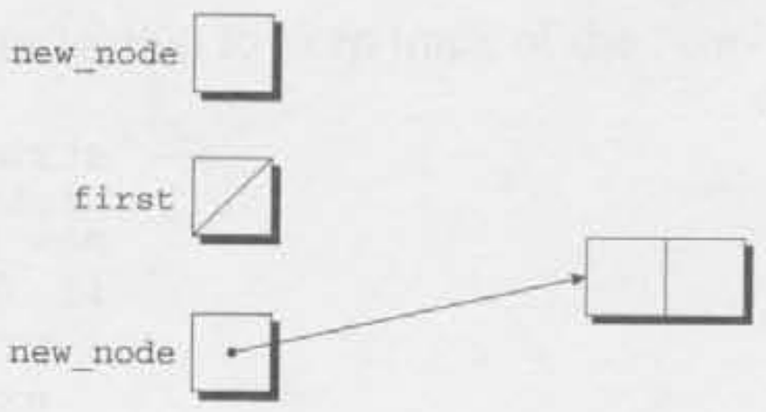
Second, we'll make `first` point to the new node:

```
first = new_node;
```

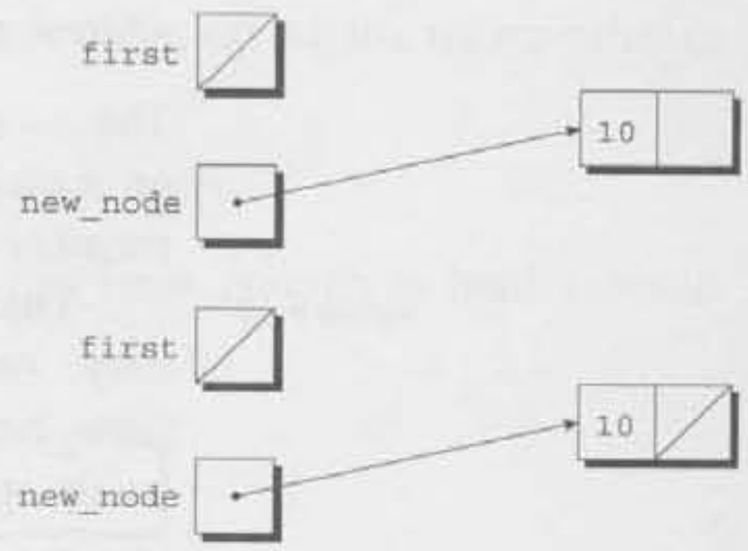
Will these statements work if the list is empty when we insert a node? Yes, fortunately. To make sure this is true, let's trace the process of inserting two nodes into an empty list. We'll insert a node containing the number 10 first, followed by a node containing 20. In the figures that follow, null pointers are shown as diagonal lines.

```
first = NULL;
```


```
new_node = malloc(sizeof(struct node));
```



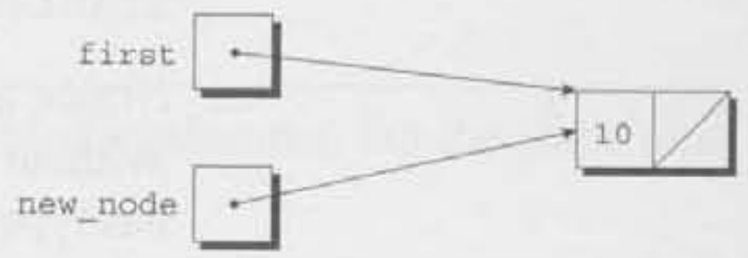
```
new_node->value = 10;
```



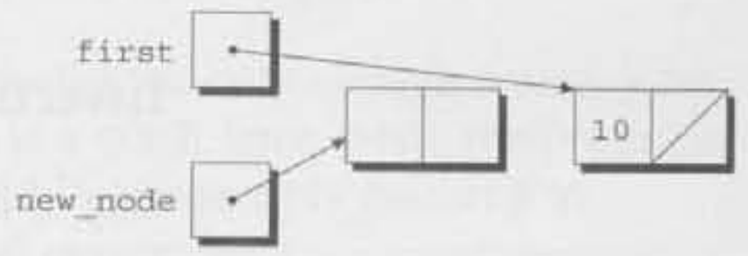

```
new_node->next = first;
```



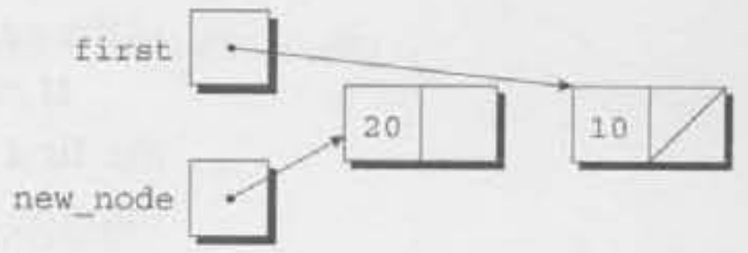
```
first = new_node;
```



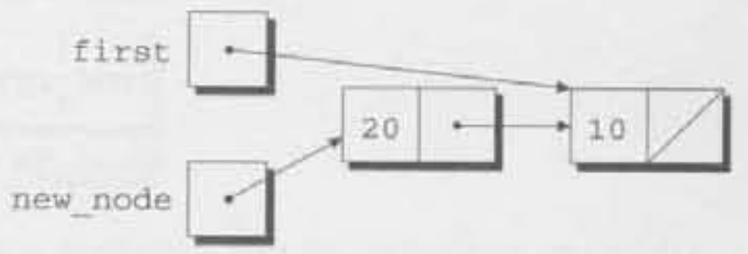

```
new_node = malloc(sizeof(struct node));
```



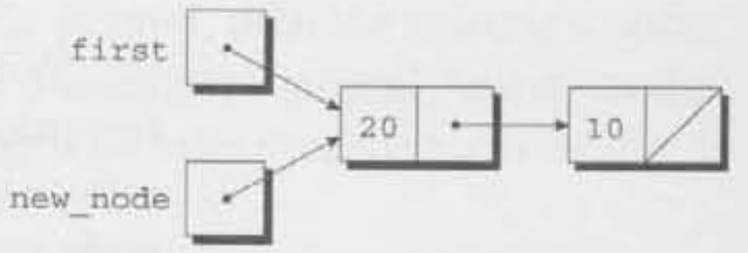

```
new_node->value = 20;
```




```
new_node->next = first;
```




```
first = new_node;
```



Inserting a node into a linked list is such a common operation that we'll probably want to write a function for that purpose. Let's name the function `add_to_list`. It will have two parameters: `list` (a pointer to the first node in the old list) and `n` (the integer to be stored in the new node).

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
```

```
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Note that `add_to_list` doesn't modify the `list` pointer. Instead, it returns a pointer to the newly created node (now at the beginning of the list). When we call `add_to_list`, we'll need to store its return value into `first`:

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

These statements add nodes containing 10 and 20 to the list pointed to by `first`. Getting `add_to_list` to update `first` directly, rather than return a new value for `first`, turns out to be tricky. We'll return to this issue in Section 17.6.

The following function uses `add_to_list` to create a linked list containing numbers entered by the user:

```
struct node *read_numbers(void)
{
  struct node *first = NULL;
  int n;

  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n == 0)
      return first;
    first = add_to_list(first, n);
  }
}
```

The numbers will be in reverse order within the list, since `first` always points to the node containing the last number entered.

Searching a Linked List

Once we've created a linked list, we may need to search it for a particular piece of data. Although a `while` loop can be used to search a list, the `for` statement is often superior. We're accustomed to using the `for` statement when writing loops that involve counting, but its flexibility makes the `for` statement suitable for other tasks as well, including operations on linked lists. Here's the customary way to visit the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node:

idiom

```
for (p = first; p != NULL; p = p->next)
  ...
```

The assignment

```
p = p->next
```

advances the p pointer from one node to the next. An assignment of this form is invariably used in C when writing a loop that traverses a linked list.

Let's write a function named search_list that searches a list (pointed to by the parameter list) for an integer n. If it finds n, search_list will return a pointer to the node containing n; otherwise, it will return a null pointer. Our first version of search_list relies on the "list-traversal" idiom:

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

Of course, there are many other ways to write search_list. One alternative would be to eliminate the p variable, instead using list itself to keep track of the current node:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

Since list is a copy of the original list pointer, there's no harm in changing it within the function.

Another alternative is to combine the list->value == n test with the list != NULL test:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL && list->value != n; list = list->next)
    ;
  return list;
}
```

Since list is NULL if we reach the end of the list, returning list is correct even if we don't find n. This version of search_list might be a bit clearer if we used a while statement:

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

Deleting a Node from a Linked List

A big advantage of storing data in a linked list is that we can easily delete nodes that we no longer need. Deleting a node, like creating a node, involves three steps:

1. Locate the node to be deleted.
2. Alter the previous node so that it "bypasses" the deleted node.
3. Call `free` to reclaim the space occupied by the deleted node.

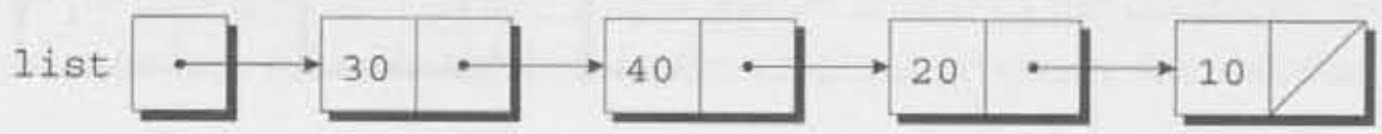
Step 1 is harder than it looks. If we search the list in the obvious way, we'll end up with a pointer to the node to be deleted. Unfortunately, we won't be able to perform step 2, which requires changing the *previous* node.

There are various solutions to this problem. We'll use the "trailing pointer" technique: as we search the list in step 1, we'll keep a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`). If `list` points to the list to be searched and `n` is the integer to be deleted, the following loop implements step 1:

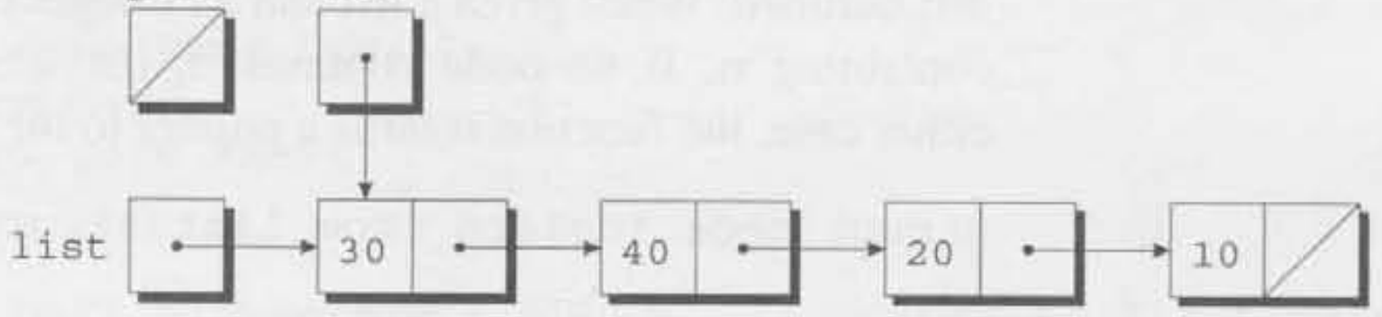
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

Here we see the power of C's `for` statement. This rather exotic example, with its empty body and liberal use of the comma operator, performs all the actions needed to search for `n`. When the loop terminates, `cur` points to the node to be deleted, while `prev` points to the previous node (if there is one).

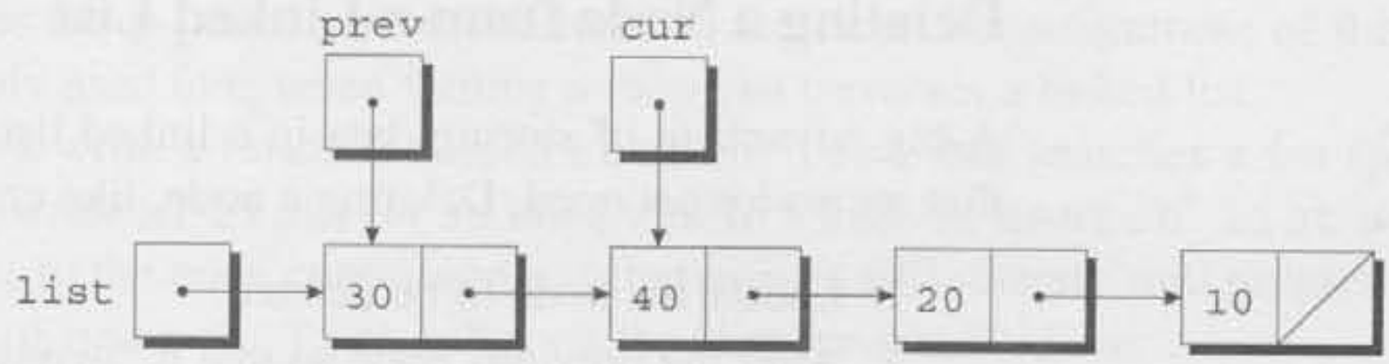
To see how this loop works, let's assume that `list` points to a list containing 30, 40, 20, and 10, in that order:



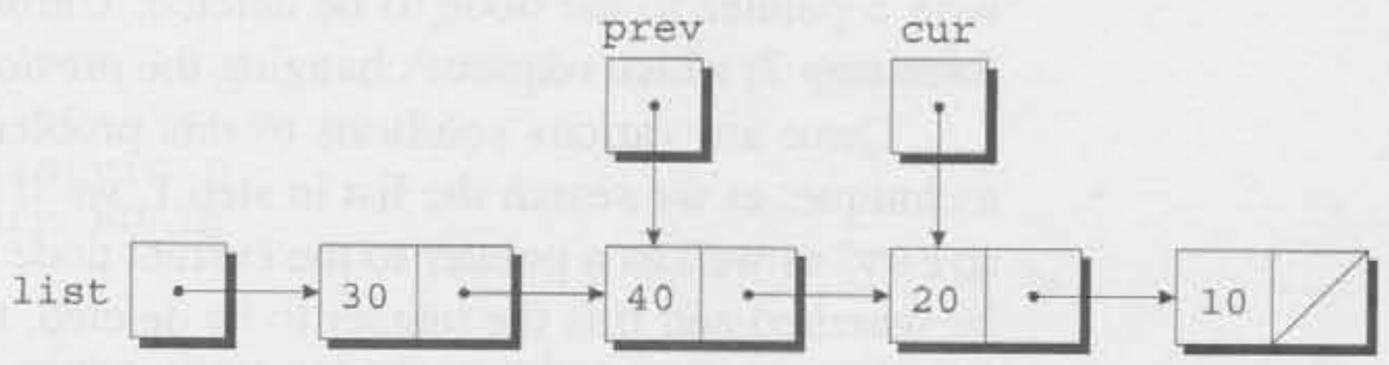
Let's say that `n` is 20, so our goal is to delete the third node in the list. After `cur = list, prev = NULL` has been executed, `cur` points to the first node in the list:



The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20. After `prev = cur, cur = cur->next` has been executed, we begin to see how the `prev` pointer will trail behind `cur`:



Again, the test `cur != NULL && cur->value != n` is true, so `prev = cur, cur = cur->next` is executed once more:

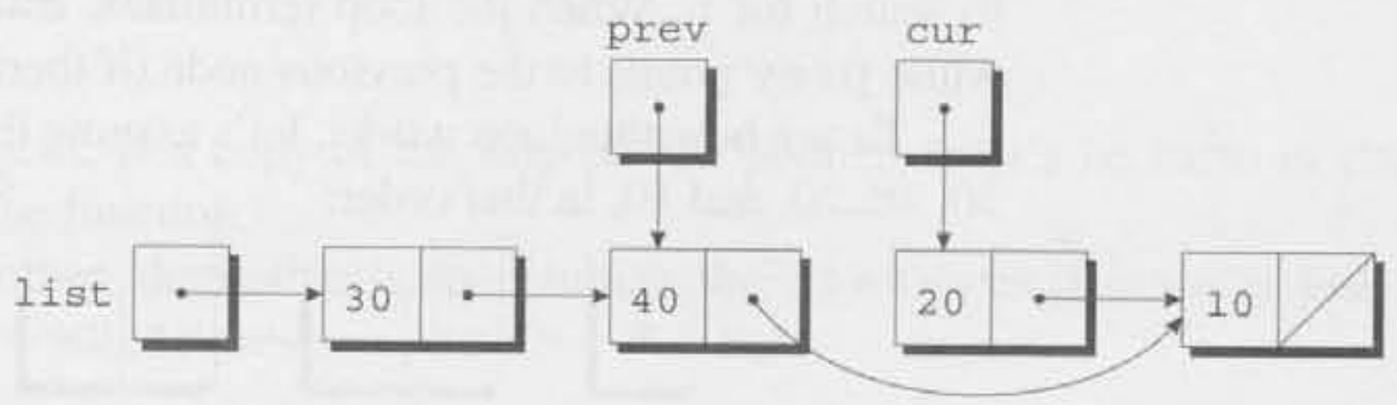


Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

Next, we'll perform the bypass required by step 2. The statement

```
prev->next = cur->next;
```

makes the pointer in the previous node point to the node *after* the current node:



We're now ready for step 3, releasing the memory occupied by the current node:

```
free(cur);
```

The following function, `delete_from_list`, uses the strategy that we've just outlined. When given a list and an integer `n`, the function deletes the first node containing `n`. If no node contains `n`, `delete_from_list` does nothing. In either case, the function returns a pointer to the list.

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)
    ;
```

```
    if (cur == NULL)
      return list;                  /* n was not found */
    if (prev == NULL)
      list = list->next;            /* n is in the first node */
    else
      prev->next = cur->next;       /* n is in some other node */
    free(cur);
    return list;
}
```

Deleting the first node in the list is a special case. The `prev == NULL` test checks for this case, which requires a different bypass step.

Ordered Lists

When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is ***ordered.*** Inserting a node into an ordered list is more difficult (the node won't always be put at the beginning of the list), but searching is faster (we can stop looking after reaching the point at which the desired node would have been located). The following program illustrates both the increased difficulty of inserting a node and the faster search.

PROGRAM

Maintaining a Parts Database (Revisited)

Let's redo the parts database program of Section 16.3, this time storing the database in a linked list. Using a linked list instead of an array has two major advantages: (1) We don't need to put a preset limit on the size of the database; it can grow until there's no more memory to store parts. (2) We can easily keep the database sorted by part number—when a new part is added to the database, we simply insert it in its proper place in the list. In the original program, the database wasn't sorted.

In the new program, the `part` structure will contain an additional member (a pointer to the next node in the linked list), and the variable `inventory` will be a pointer to the first node in the list:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;    /* points to first part */
```

Most of the functions in the new program will closely resemble their counterparts in the original program. The `find_part` and `insert` functions will be more complex, however, since we'll keep the nodes in the `inventory` list sorted by part number.

In the original program, `find_part` returns an index into the `inventory` array. In the new program, `find_part` will return a pointer to the node that contains the desired part number. If it doesn't find the part number, `find_part` will return a null pointer. Since the `inventory` list is sorted by part number, the new version of `find_part` can save time by stopping its search when it finds a node containing a part number that's greater than or equal to the desired part number. `find_part`'s search loop will have the form

```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)
  ;
```

The loop will terminate when `p` becomes `NULL` (indicating that the part number wasn't found) or when `number > p->number` is false (indicating that the part number we're looking for is less than or equal to a number already stored in a node). In the latter case, we still don't know whether or not the desired number is actually in the list, so we'll need another test:

```
if (p != NULL && number == p->number)
  return p;
```

The original version of `insert` stores a new part in the next available array element. The new version must determine where the new part belongs in the list and insert it there. We'll also have `insert` check whether the part number is already present in the list. `insert` can accomplish both tasks by using a loop similar to the one in `find_part`:

```
for (cur = inventory, prev = NULL;
     cur != NULL && new_node->number > cur->number;
     prev = cur, cur = cur->next)
  ;
```

This loop relies on two pointers: `cur`, which points to the current node, and `prev`, which points to the previous node. Once the loop terminates, `insert` will check whether `cur` isn't `NULL` and `new_node->number` equals `cur->number`; if so, the part number is already in the list. Otherwise `insert` will insert a new node between the nodes pointed to by `prev` and `cur`, using a strategy similar to the one we employed for deleting a node. (This strategy works even if the new part number is larger than any in the list; in that case, `cur` will be `NULL` but `prev` will point to the last node in the list.)

Here's the new program. Like the original program, this version requires the `read_line` function described in Section 16.3; I assume that `readline.h` contains a prototype for this function.

inventory2.c

```
/* Maintains a parts database (linked list version) */

#include <stdio.h>
#include <stdlib.h>
#include "readline.h"
```

```
#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;   /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
char code;

for (;;) {
  printf("Enter operation code: ");
  scanf(" %c", &code);
  while (getchar() != '\n')   /* skips to end of line */
    ;
  switch (code) {
    case 'i': insert();
              break;
    case 's': search();
              break;
    case 'u': update();
              break;
    case 'p': print();
              break;
    case 'q': return 0;
    default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}

/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            list. Returns a pointer to the node         *
 *            containing the part number; if the part     *
 *            number is not found, returns NULL.          *
 **********************************************************/
```

```
struct part *find_part(int number)
{
  struct part *p;

  for (p = inventory;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
  return NULL;
}

/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         inventory list; the list remains sorted by     *
 *         part number. Prints an error message and       *
 *         returns prematurely if the part already exists *
 *         or space could not be allocated for the part.  *
 **********************************************************/
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);

  for (cur = inventory, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;
  if (cur != NULL && new_node->number == cur->number) {
    printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);

  new_node->next = cur;
  if (prev == NULL)
    inventory = new_node;
  else
    prev->next = new_node;
}
```

```
/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}

/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
    printf("Part not found.\n");
}

/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Part numbers will appear in   *
 *        ascending order.                                *
 **********************************************************/
void print(void)
{
  struct part *p;
```

```
    printf("Part Number   Part Name                      "
           "Quantity on Hand\n");
    for (p = inventory; p != NULL; p = p->next)
      printf("%7d       %-25s%11d\n", p->number, p->name,
             p->on_hand);
}
```

Notice the use of `free` in the `insert` function. `insert` allocates memory for a part before checking to see if the part already exists. If it does, `insert` releases the space to avoid a memory leak.

17.6 Pointers to Pointers

In Section 13.7, we came across the notion of a *pointer* to a *pointer*. In that section, we used an array whose elements were of type `char *`; a pointer to one of the array elements itself had type `char **`. The concept of "pointers to pointers" also pops up frequently in the context of linked data structures. In particular, when an argument to a function is a pointer variable, we'll sometimes want the function to be able to modify the variable by making it point somewhere else. Doing so requires the use of a pointer to a pointer.

Consider the `add_to_list` function of Section 17.5, which inserts a node at the beginning of a linked list. When we call `add_to_list`, we pass it a pointer to the first node in the original list; it then returns a pointer to the first node in the updated list:

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Suppose that we modify the function so that it assigns `new_node` to `list` instead of returning `new_node`. In other words, let's remove the `return` statement from `add_to_list` and replace it by

```
list = new_node;
```

Unfortunately, this idea doesn't work. Suppose that we call `add_to_list` in the following way:

```
add_to_list(first, 10);
```

At the point of the call, `first` is copied into `list`. (Pointers, like all arguments, are passed by value.) The last line in the function changes the value of `list`, making it point to the new node. This assignment doesn't affect `first`, however.

Getting `add_to_list` to modify `first` is possible, but it requires passing `add_to_list` a *pointer* to `first`. Here's the correct version of the function:

```
void add_to_list(struct node **list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = *list;
  *list = new_node;
}
```

When we call the new version of `add_to_list`, the first argument will be the address of `first`:

```
add_to_list(&first, 10);
```

Since `list` is assigned the address of `first`, we can use `*list` as an alias for `first`. In particular, assigning `new_node` to `*list` will modify `first`.

17.7 Pointers to Functions

We've seen that pointers may point to various kinds of data, including variables, array elements, and dynamically allocated blocks of memory. But C doesn't require that pointers point only to *data;* it's also possible to have pointers to *functions.* Pointers to functions aren't as odd as you might think. After all, functions occupy memory locations, so every function has an address, just as each variable has an address.

Function Pointers as Arguments

We can use function pointers in much the same way we use pointers to data. In particular, passing a function pointer as an argument is fairly common in C. Suppose that we're writing a function named `integrate` that integrates a mathematical function `f` between points `a` and `b`. We'd like to make `integrate` as general as possible by passing it `f` as an argument. To achieve this effect in C, we'll declare `f` to be a pointer to a function. Assuming that we want to integrate functions that have

a `double` parameter and return a `double` result, the prototype for `integrate` will look like this:

```
double integrate(double (*f)(double), double a, double b);
```

The parentheses around `*f` indicate that `f` is a pointer to a function, not a function that returns a pointer. It's also legal to declare `f` as though it were a function:

```
double integrate(double f(double), double a, double b);
```

From the compiler's standpoint, this prototype is identical to the previous one.

When we call `integrate`, we'll supply a function name as the first argument. For example, the following call will integrate the `sin` (sine) function from 0 to π/2:

sin function ➤23.3

```
result = integrate(sin, 0.0, PI / 2);
```

Notice that there are no parentheses after `sin`. When a function name isn't followed by parentheses, the C compiler produces a pointer to the function instead of generating code for a function call. In our example, we're not calling `sin`; instead, we're passing `integrate` a pointer to `sin`. If this seems confusing, think of how C handles arrays. If `a` is the name of an array, then `a[i]` represents one element of the array, while `a` by itself serves as a pointer to the array. In a similar way, if `f` is a function, C treats `f(x)` as a *call* of the function but `f` by itself as a *pointer* to the function.

Within the body of `integrate`, we can call the function that `f` points to:

```
y = (*f)(x);
```

`*f` represents the function that `f` points to; `x` is the argument to the call. Thus, during the execution of `integrate(sin, 0.0, PI / 2)`, each call of `*f` is actually a call of `sin`. As an alternative to `(*f)(x)`, C allows us to write `f(x)` to call the function that `f` points to. Although `f(x)` looks more natural, I'll stick with `(*f)(x)` as a reminder that `f` is a pointer to a function, not a function name.

The `qsort` Function

Although it might seem that pointers to functions aren't relevant to the average programmer, that couldn't be further from the truth. In fact, some of the most useful functions in the C library require a function pointer as an argument. One of these is `qsort`, which belongs to the `<stdlib.h>` header. `qsort` is a general-purpose sorting function that's capable of sorting any array, based on any criteria that we choose.

Q&A

Since the elements of the array that it sorts may be of any type—even a structure or union type—`qsort` must be told how to determine which of two array elements is "smaller." We'll provide this information to `qsort` by writing a ***comparison function.*** When given two pointers `p` and `q` to array elements, the comparison function must return an integer that is *negative* if `*p` is "less than" `*q`,

zero if *p is "equal to" *q, and *positive* if *p is "greater than" *q. The terms "less than," "equal to," and "greater than" are in quotes because it's our responsibility to determine how *p and *q are compared.

qsort has the following prototype:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

base must point to the first element in the array. (If only a portion of the array is to be sorted, we'll make base point to the first element in this portion.) In the simplest case, base is just the name of the array. nmemb is the number of elements to be sorted (not necessarily the number of elements in the array). size is the size of each array element, measured in bytes. compar is a pointer to the comparison function. When qsort is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

To sort the inventory array of Section 16.3, we'd use the following call of qsort:

Q&A

```
qsort(inventory, num_parts, sizeof(struct part), compare_parts);
```

Notice that the second argument is num_parts, not MAX_PARTS; we don't want to sort the entire inventory array, just the portion in which parts are currently stored. The last argument, compare_parts, is a function that compares two part structures.

Writing the compare_parts function isn't as easy as you might expect. qsort requires that its parameters have type void *, but we can't access the members of a part structure through a void * pointer; we need a pointer of type struct part * instead. To solve the problem, we'll have compare_parts assign its parameters, p and q, to variables of type struct part *, thereby converting them to the desired type. compare_parts can now use these variables to access the members of the structures that p and q point to. Assuming that we want to sort the inventory array into ascending order by part number, here's how the compare_parts function might look:

```
int compare_parts(const void *p, const void *q)
{
  const struct part *p1 = p;
  const struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

The declarations of p1 and q1 include the word const to avoid getting a warning from the compiler. Since p and q are const pointers (indicating that the objects

to which they point should not be modified), they should be assigned only to pointer variables that are also declared to be `const`.

Although this version of `compare_parts` works, most C programmers would write the function more concisely. First, notice that we can replace `p1` and `q1` by cast expressions:

```
int compare_parts(const void *p, const void *q)
{
  if (((struct part *) p)->number <
      ((struct part *) q)->number)
    return -1;
  else if (((struct part *) p)->number ==
           ((struct part *) q)->number)
    return 0;
  else
    return 1;
}
```

The parentheses around `((struct part *) p)` are necessary; without them, the compiler would try to cast `p->number` to type `struct part *`.

We can make `compare_parts` even shorter by removing the `if` statements:

```
int compare_parts(const void *p, const void *q)
{
  return ((struct part *) p)->number -
         ((struct part *) q)->number;
}
```

Subtracting `q`'s part number from `p`'s part number produces a negative result if `p` has a smaller part number, zero if the part numbers are equal, and a positive result if `p` has a larger part number. (Note that subtracting two integers is potentially risky because of the danger of overflow. I'm assuming that part numbers are positive integers, so that shouldn't happen here.)

To sort the `inventory` array by part name instead of part number, we'd use the following version of `compare_parts`:

```
int compare_parts(const void *p, const void *q)
{
  return strcmp(((struct part *) p)->name,
                ((struct part *) q)->name);
}
```

All `compare_parts` has to do is call `strcmp`, which conveniently returns a negative, zero, or positive result.

Other Uses of Function Pointers

Although I've emphasized the usefulness of function pointers as arguments to other functions, that's not all they're good for. C treats pointers to functions just like pointers to data; we can store function pointers in variables or use them as ele-

ments of an array or as members of a structure or union. We can even write functions that return function pointers.

Here's an example of a variable that can store a pointer to a function:

```
void (*pf)(int);
```

`pf` can point to any function with an `int` parameter and a return type of `void`. If `f` is such a function, we can make `pf` point to `f` in the following way:

```
pf = f;
```

Notice that there's no ampersand preceding `f`. Once `pf` points to `f`, we can call `f` by writing either

```
(*pf)(i);
```

or

```
pf(i);
```

Arrays whose elements are function pointers have a surprising number of applications. For example, suppose that we're writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, then store pointers to the functions in an array:

```
void (*file_cmd[])(void) = {new_cmd,
                            open_cmd,
                            close_cmd,
                            close_all_cmd,
                            save_cmd,
                            save_as_cmd,
                            save_all_cmd,
                            print_cmd,
                            exit_cmd
                           };
```

If the user selects command `n`, where `n` falls between 0 and 8, we can subscript the `file_cmd` array and call the corresponding function:

```
(*file_cmd[n])();    /* or file_cmd[n](); */
```

Of course, we could get a similar effect with a `switch` statement. Using an array of function pointers gives us more flexibility, however, since the elements of the array can be changed as the program is running.

PROGRAM Tabulating the Trigonometric Functions

The following program prints tables showing the values of the `cos`, `sin`, and `tan` functions (all three belong to `<math.h>`). The program is built around a function named `tabulate` that, when passed a function pointer `f`, prints a table showing the values of `f`.

`<math.h>` header ➤23.3

tabulate.c

```
/* Tabulates values of trigonometric functions */

#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first,
              double last, double incr);

int main(void)
{
  double final, increment, initial;

  printf("Enter initial value: ");
  scanf("%lf", &initial);

  printf("Enter final value: ");
  scanf("%lf", &final);

  printf("Enter increment: ");
  scanf("%lf", &increment);

  printf("\n      x         cos(x)"
         "\n   -------    --------\n");
  tabulate(cos, initial, final, increment);

  printf("\n      x         sin(x)"
         "\n   -------    --------\n");
  tabulate(sin, initial, final, increment);

  printf("\n      x         tan(x)"
         "\n   -------    --------\n");
  tabulate(tan, initial, final, increment);

  return 0;
}

void tabulate(double (*f)(double), double first,
              double last, double incr)
{
  double x;
  int i, num_intervals;

  num_intervals = ceil((last - first) / incr);
  for (i = 0; i <= num_intervals; i++) {
    x = first + i * incr;
    printf("%10.5f %10.5f\n", x, (*f)(x));
  }
}
```

`tabulate` uses the `ceil` function, which also in `<math.h>`. When given an argument `x` of `double` type, `ceil` returns the smallest integer that's greater than or equal to `x`.

Here's what a session with `tabulate.c` might look like:

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1

      x         cos(x)
-------     -------
0.00000     1.00000
0.10000     0.99500
0.20000     0.98007
0.30000     0.95534
0.40000     0.92106
0.50000     0.87758

      x         sin(x)
-------     -------
0.00000     0.00000
0.10000     0.09983
0.20000     0.19867
0.30000     0.29552
0.40000     0.38942
0.50000     0.47943

      x         tan(x)
-------     -------
0.00000     0.00000
0.10000     0.10033
0.20000     0.20271
0.30000     0.30934
0.40000     0.42279
0.50000     0.54630
```

17.8 Restricted Pointers (C99)

This section and the next discuss two of C99's pointer-related features. Both are primarily of interest to advanced C programmers; most readers will want to skip these sections.

In C99, the keyword `restrict` may appear in the declaration of a pointer:

```
int * restrict p;
```

A pointer that's been declared using `restrict` is called a ***restricted pointer.*** The intent is that if `p` points to an object that is later modified, then that object is not accessed in any way other than through `p`. (Alternative ways to access the object include having another pointer to the same object or having `p` point to a named variable.) Having more than one way to access an object is often called ***aliasing.***

Let's look at an example of the kind of behavior that restricted pointers are supposed to discourage. Suppose that `p` and `q` have been declared as follows:

```
int * restrict p;
int * restrict q;
```

Now suppose that p is made to point to a dynamically allocated block of memory:

```
p = malloc(sizeof(int));
```

(A similar situation would arise if p were assigned the address of a variable or an array element.) Normally it would be legal to copy p into q and then modify the integer through q:

```
q = p;
*q = 0;    /* causes undefined behavior */
```

Because p is a restricted pointer, however, the effect of executing the statement *q = 0; is undefined. By making p and q point to the same object, we caused *p and *q to be aliases.

extern storage class ➤ 18.2

If a restricted pointer p is declared as a local variable without the extern storage class, restrict applies only to p when the block in which p is declared is being executed. (Note that the body of a function is a block.) restrict can be used with function parameters of pointer type, in which case it applies only when the function is executing. When restrict is applied to a pointer variable with file scope, however, the restriction lasts for the entire execution of the program.

blocks ➤ 10.3

file scope ➤ 10.2

The exact rules for using restrict are rather complex; see the C99 standard for details. There are even situations in which an alias created from a restricted pointer is legal. For example, a restricted pointer p can be legally copied into another restricted pointer variable q, provided that p is local to a function and q is defined inside a block nested within the function's body.

To illustrate the use of restrict, let's look at the memcpy and memmove functions, which belong to the <string.h> header. memcpy has the following prototype in C99:

<string.h> header ➤ 23.6

```
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
```

memcpy is similar to strcpy, except that it copies bytes from one object to another (strcpy copies characters from one string into another). s2 points to the data to be copied, s1 points to the destination of the copy, and n is the number of bytes to be copied. The use of restrict with both s1 and s2 indicates that the source of the copy and the destination shouldn't overlap. (It doesn't *guarantee* that they don't overlap, however.)

In contrast, restrict doesn't appear in the prototype for memmove:

```
void *memmove(void *s1, const void *s2, size_t n);
```

memmove does the same thing as memcpy: it copies bytes from one place to another. The difference is that memmove is guaranteed to work even if the source and destination overlap. For example, we could use memmove to shift the elements of an array by one position:

```
int a[100];
...
```

```
memmove(&a[0], &a[1], 99 * sizeof(int));
```

Prior to C99, there was no way to document the difference between memcpy and memmove. The prototypes for the two functions were nearly identical:

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

The use of `restrict` in the C99 version of memcpy's prototype lets the programmer know that `s1` and `s2` should point to objects that don't overlap, or else the function isn't guaranteed to work.

Although using `restrict` in function prototypes is useful documentation, that's not the primary reason for its existence. `restrict` provides information to the compiler that may enable it to produce more efficient code—a process known as ***optimization***. (The `register` storage class serves the same purpose.) Not every compiler attempts to optimize programs, however, and the ones that do normally allow the programmer to disable optimization. As a result, the C99 standard guarantees that `restrict` has no effect on the behavior of a program that conforms to the standard: if all uses of `restrict` are removed from such a program, it should behave the same.

register storage class ➤ 18.2

Most programmers won't use `restrict` unless they're fine-tuning a program to achieve the best possible performance. Still, it's worth knowing about `restrict` because it appears in the C99 prototypes for a number of standard library functions.

17.9 Flexible Array Members (C99)

Every once in a while, we'll need to define a structure that contains an array of an unknown size. For example, we might want to store strings in a form that's different from the usual one. Normally, a string is an array of characters, with a null character marking the end. However, there are advantages to storing strings in other ways. One alternative is to store the length of the string along with the string's characters (but with no null character). The length and the characters could be stored in a structure such as this one:

```
struct vstring {
  int len;
  char chars[N];
};
```

Here `N` is a macro that represents the maximum length of a string. Using a fixed-length array such as this is undesirable, however, because it forces us to limit the length of the string, plus it wastes memory (since most strings won't need all `N` characters in the array).

C programmers have traditionally solved this problem by declaring the length of `chars` to be 1 (a dummy value) and then dynamically allocating each string:

```
struct vstring {
  int len;
  char chars[1];
};
...
struct vstring *str = malloc(sizeof(struct vstring) + n - 1);
str->len = n;
```

We're "cheating" by allocating more memory than the structure is declared to have (in this case, an extra `n` – 1 characters), and then using the memory to store additional elements of the `chars` array. This technique has become so common over the years that it has a name: the "struct hack."

The struct hack isn't limited to character arrays; it has a variety of uses. Over time, it has become popular enough to be supported by many compilers. Some (including GCC) even allow the `chars` array to have zero length, which makes this trick a little more explicit. Unfortunately, the C89 standard doesn't guarantee that the struct hack will work, nor does it allow zero-length arrays.

In recognition of the struct hack's usefulness, C99 has a feature known as the ***flexible array member*** that serves the same purpose. When the last member of a structure is an array, its length may be omitted:

```
struct vstring {
  int len;
  char chars[];   /* flexible array member - C99 only */
};
```

The length of the `chars` array isn't determined until memory is allocated for a `vstring` structure, normally using a call of `malloc`:

```
struct vstring *str = malloc(sizeof(struct vstring) + n);
str->len = n;
```

In this example, `str` points to a `vstring` structure in which the `chars` array occupies `n` characters. The `sizeof` operator ignores the `chars` member when computing the size of the structure. (A flexible array member is unusual in that it takes up no space within a structure.)

A few special rules apply to a structure that contains a flexible array member. The flexible array member must appear last in the structure, and the structure must have at least one other member. Copying a structure that contains a flexible array member will copy the other members but not the flexible array itself.

A structure that contains a flexible array member is an ***incomplete type.*** An incomplete type is missing part of the information needed to determine how much memory it requires. Incomplete types, which are discussed further in one of the Q&A questions at the end of this chapter and in Section 19.3, are subject to various restrictions. In particular, an incomplete type (and hence a structure that contains a flexible array member) can't be a member of another structure or an element of an array. However, an array may contain pointers to structures that have a flexible array member; Programming Project 7 at the end of this chapter is built around such an array.

Q & A

Q: What does the `NULL` macro represent? [p. 415]

A: `NULL` actually stands for `0`. When we use `0` in a context where a pointer would be required, C compilers treat it as a null pointer instead of the integer 0. The `NULL` macro is provided merely to help avoid confusion. The assignment

```
p = 0;
```

could be assigning the value 0 to a numeric variable or assigning a null pointer to a pointer variable; we can't easily tell which. In contrast, the assignment

```
p = NULL;
```

makes it clear that `p` is a pointer.

***Q: In the header files that come with my compiler, `NULL` is defined as follows:**

```
#define NULL (void *) 0
```

What's the advantage of casting `0` to `void *`?

A: This trick, which is allowed by the C standard, enables compilers to spot incorrect uses of the null pointer. For example, suppose that we try to assign `NULL` to an integer variable:

```
i = NULL;
```

If `NULL` is defined as `0`, this assignment is perfectly legal. But if `NULL` is defined as `(void *) 0`, the compiler can warn us that we're assigning a pointer to an integer variable.

variable-length argument lists ➤26.1

Defining `NULL` as `(void *) 0` has a second, more important, advantage. Suppose that we call a function with a variable-length argument list and pass `NULL` as one of the arguments. If `NULL` is defined as `0`, the compiler will incorrectly pass a zero integer value. (In an ordinary function call, `NULL` works fine because the compiler knows from the function's prototype that it expects a pointer. When a function has a variable-length argument list, however, the compiler lacks this knowledge.) If `NULL` is defined as `(void *) 0`, the compiler will pass a null pointer.

To make matters even more confusing, some header files define `NULL` to be `0L` (the `long` version of 0). This definition, like the definition of `NULL` as `0`, is a holdover from C's earlier years, when pointers and integers were compatible. For most purposes, though, it really doesn't matter how `NULL` is defined; just think of it as a name for the null pointer.

Q: Since `0` is used to represent the null pointer, I guess a null pointer is just an address with all zero bits, right?

A: Not necessarily. Each C compiler is allowed to represent null pointers in a different way, and not all compilers use a zero address. For example, some compilers use a nonexistent memory address for the null pointer; that way, attempting to access memory through a null pointer can be detected by the hardware.

How the null pointer is stored inside the computer shouldn't concern us; that's a detail for compiler experts to worry about. The important thing is that, when used in a pointer context, `0` is converted to the proper internal form by the compiler.

Q: Is it acceptable to use `NULL` as a null character?

A: Definitely not. `NULL` is a macro that represents the null *pointer*, not the null *character*. Using `NULL` as a null character will work with some compilers, but not with all (since some define `NULL` as `(void *) 0`). In any event, using `NULL` as anything other than a pointer can lead to a great deal of confusion. If you want a name for the null character, define the following macro:

```
#define NUL '\0'
```

***Q: When my program terminates, I get the message *"Null pointer assignment."* What does this mean?**

A: This message, which is produced by programs compiled with some older DOS-based C compilers, indicates that the program has stored data in memory using a bad pointer (but not necessarily a null pointer). Unfortunately, the message isn't displayed until the program terminates, so there's no clue as to which statement caused the error. The *"Null pointer assignment"* message can be caused by a missing `&` in `scanf`:

```
scanf("%d", i);    /* should have been scanf("%d", &i); */
```

Another possibility is an assignment involving a pointer that's uninitialized or null:

```
*p = i;    /* p is uninitialized or null */
```

***Q: How does a program know that a "null pointer assignment" has occurred?**

A: The message depends on the fact that, in the small and medium memory models, data is stored in a single segment, with addresses beginning at 0. The compiler leaves a "hole" at the beginning of the data segment—a small block of memory that's initialized to 0 but otherwise isn't used by the program. When the program terminates, it checks to see if any data in the "hole" area is nonzero. If so, it must have been altered through a bad pointer.

Q: Is there any advantage to casting the return value of `malloc` or the other memory allocation functions? [p. 416]

A: Not usually. Casting the `void *` pointer that these functions return is unnecessary, since pointers of type `void *` are automatically converted to any pointer type upon assignment. The habit of casting the return value is a holdover from older versions of C, in which the memory allocation functions returned a `char *` value, making the cast necessary. Programs that are designed to be compiled as C++ code

may benefit from the cast, but that's about the only reason to do it.

In C89, there's actually a small advantage to *not* performing the cast. Suppose that we've forgotten to include the `<stdlib.h>` header in our program. When we call `malloc`, the compiler will assume that its return type is `int` (the default return value for any C function). If we don't cast the return value of `malloc`, a C89 compiler will produce an error (or at least a warning), since we're trying to assign an integer value to a pointer variable. On the other hand, if we cast the return value to a pointer, the program may compile, but likely won't run properly. With C99, this advantage disappears. Forgetting to include the `<stdlib.h>` header will cause an error when `malloc` is called, because C99 requires that a function be declared before it's called.

Q: The `calloc` function initializes a memory block by setting its bits to zero. Does this mean that all data items in the block become zero? [p. 421]

A: Usually, but not always. Setting an integer to zero bits always makes the integer zero. Setting a floating-point number to zero bits usually makes the number zero, but this isn't guaranteed—it depends on how floating-point numbers are stored. The story is the same for pointers; a pointer whose bits are zero isn't necessary a null pointer.

***Q: I see how the structure tag mechanism allows a structure to contain a pointer to itself. But what if two structures each have a member that points to the other? [p. 425]**

A: Here's how we'd handle that situation:

```
struct s1;   /* incomplete declaration of s1 */

struct s2 {
  …
  struct s1 *p;
  …
};

struct s1 {
  …
  struct s2 *q;
  …
};
```

incomplete types ➤ *19.3*

The first declaration of `s1` creates an incomplete structure type, since we haven't specified the members of `s1`. The second declaration of `s1` "completes" the type by describing the members of the structure. Incomplete declarations of a structure type are permitted in C, although their uses are limited. Creating a pointer to such a type (as we did when declaring `p`) is one of these uses.

Q: Calling `malloc` with the wrong argument—causing it to allocate too much memory or too little memory—seems to be a common error. Is there a safer way to use `malloc`? [p. 426]

A: Yes, there is. Some programmers use the following idiom when calling `malloc` to allocate memory for a single object:

```
p = malloc(sizeof(*p));
```

Since `sizeof(*p)` is the size of the object to which `p` will point, this statement guarantees that the correct amount of memory will be allocated. At first glance, this idiom looks fishy: it's likely that `p` is uninitialized, making the value of `*p` undefined. However, `sizeof` doesn't evaluate `*p`, it merely computes its size, so the idiom works even if `p` is uninitialized or contains a null pointer.

To allocate memory for an array with `n` elements, we can use a slightly modified version of the idiom:

```
p = malloc(n * sizeof(*p));
```

Q: Why isn't the `qsort` function simply named `sort`? [p. 440]

A: The name `qsort` comes from the Quicksort algorithm published by C. A. R. Hoare in 1962 (and discussed in Section 9.6). Ironically, the C standard doesn't require that `qsort` use the Quicksort algorithm, although many versions of `qsort` do.

Q: Isn't it necessary to cast `qsort`'s first argument to type `void *`, as in the following example? [p. 441]

```
qsort((void *) inventory, num_parts, sizeof(struct part),
      compare_parts);
```

A: No. A pointer of any type can be converted to `void *` automatically.

***Q: I want to use `qsort` to sort an array of integers, but I'm having trouble writing a comparison function. What's the secret?**

A: Here's a version that works:

```
int compare_ints(const void *p, const void *q)
{
  return *(int *)p - *(int *)q;
}
```

Bizarre, eh? The expression `(int *)p` casts `p` to type `int *`, so `*(int *)p` would be the integer that `p` points to. A word of warning, though: Subtracting two integers may cause overflow. If the integers being sorted are completely arbitrary, it's safer to use `if` statements to compare `*(int *)p` with `*(int *)q`.

***Q: I needed to sort an array of strings, so I figured I'd just use `strcmp` as the comparison function. When I passed it to `qsort`, however, the compiler gave me a warning. I tried to fix the problem by embedding `strcmp` in a comparison function:**

```
int compare_strings(const void *p, const void *q)
{
  return strcmp(p, q);
}
```

Now my program compiles, but `qsort` doesn't seem to sort the array. What am I doing wrong?

A: First, you can't pass `strcmp` itself to `qsort`, since `qsort` requires a comparison function with two `const void *` parameters. Your `compare_strings` function doesn't work because it incorrectly assumes that `p` and `q` are strings (`char *` pointers). In fact, `p` and `q` point to array elements containing `char *` pointers. To fix `compare_strings`, we'll cast `p` and `q` to type `char **`, then use the `*` operator to remove one level of indirection:

```
int compare_strings(const void *p, const void *q)
{
  return strcmp(*(char **)p, *(char **)q);
}
```

Exercises

Section 17.1

1. Having to check the return value of `malloc` (or any other memory allocation function) each time we call it can be an annoyance. Write a function named `my_malloc` that serves as a "wrapper" for `malloc`. When we call `my_malloc` and ask it to allocate `n` bytes, it in turn calls `malloc`, tests to make sure that `malloc` doesn't return a null pointer, and then returns the pointer from `malloc`. Have `my_malloc` print an error message and terminate the program if `malloc` returns a null pointer.

Section 17.2

W 2. Write a function named `duplicate` that uses dynamic storage allocation to create a copy of a string. For example, the call

```
p = duplicate(str);
```

would allocate space for a string of the same length as `str`, copy the contents of `str` into the new string, and return a pointer to it. Have `duplicate` return a null pointer if the memory allocation fails.

Section 17.3

3. Write the following function:

```
int *create_array(int n, int initial_value);
```

The function should return a pointer to a dynamically allocated `int` array with `n` members, each of which is initialized to `initial_value`. The return value should be `NULL` if the array can't be allocated.

Section 17.5

4. Suppose that the following declarations are in effect:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
struct rectangle *p;
```

Assume that we want `p` to point to a `rectangle` structure whose upper left corner is at (10, 25) and whose lower right corner is at (20, 15). Write a series of statements that allocate such a structure and initialize it as indicated.

W 5. Suppose that `f` and `p` are declared as follows:

```
struct {
  union {
    char a, b;
    int c;
  } d;
  int e[5];
} f, *p = &f;
```

Which of the following statements are legal?

(a) `p->b = ' ';`
(b) `p->e[3] = 10;`
(c) `(*p).d.a = '*';`
(d) `p->d->c = 20;`

6. Modify the `delete_from_list` function so that it uses only one pointer variable instead of two (`cur` and `prev`).

W 7. The following loop is supposed to delete all nodes from a linked list and release the memory that they occupy. Unfortunately, the loop is incorrect. Explain what's wrong with it and show how to fix the bug.

```
for (p = first; p != NULL; p = p->next)
  free(p);
```

W 8. Section 15.2 describes a file, `stack.c`, that provides functions for storing integers in a stack. In that section, the stack was implemented as an array. Modify `stack.c` so that a stack is now stored as a linked list. Replace the `contents` and `top` variables by a single variable that points to the first node in the list (the "top" of the stack). Write the functions in `stack.c` so that they use this pointer. Remove the `is_full` function, instead having `push` return either `true` (if memory was available to create a node) or `false` (if not).

9. True or false: If `x` is a structure and `a` is a member of that structure, then `(&x)->a` is the same as `x.a`. Justify your answer.

10. Modify the `print_part` function of Section 16.2 so that its parameter is a *pointer* to a `part` structure. Use the `->` operator in your answer.

11. Write the following function:

```
int count_occurrences(struct node *list, int n);
```

The `list` parameter points to a linked list; the function should return the number of times that `n` appears in this list. Assume that the `node` structure is the one defined in Section 17.5.

12. Write the following function:

```
struct node *find_last(struct node *list, int n);
```

The `list` parameter points to a linked list. The function should return a pointer to the *last* node that contains `n`; it should return `NULL` if `n` doesn't appear in the list. Assume that the `node` structure is the one defined in Section 17.5.

13. The following function is supposed to insert a new node into its proper place in an ordered list, returning a pointer to the first node in the modified list. Unfortunately, the function

doesn't work correctly in all cases. Explain what's wrong with it and show how to fix it. Assume that the node structure is the one defined in Section 17.5.

```
struct node *insert_into_ordered_list(struct node *list,
                                      struct node *new_node)
{
  struct node *cur = list, *prev = NULL;
  while (cur->value <= new_node->value) {
    prev = cur;
    cur = cur->next;
  }
  prev->next = new_node;
  new_node->next = cur;
  return list;
}
```

Section 17.6

14. Modify the `delete_from_list` function (Section 17.5) so that its first parameter has type `struct node **` (a pointer to a pointer to the first node in a list) and its return type is `void`. `delete_from_list` must modify its first argument to point to the list after the desired node has been deleted.

Section 17.7

W 15. Show the output of the following program and explain what it does.

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

int main(void)
{
  printf("Answer: %d\n", f1(f2));
  return 0;
}

int f1(int (*f)(int))
{
  int n = 0;

  while ((*f)(n)) n++;
  return n;
}

int f2(int i)
{
  return i * i + i - 12;
}
```

16. Write the following function. The call `sum(g, i, j)` should return `g(i) + ... + g(j)`.

```
int sum(int (*f)(int), int start, int end);
```

W 17. Let `a` be an array of 100 integers. Write a call of `qsort` that sorts only the *last* 50 elements in `a`. (You don't need to write the comparison function).

18. Modify the `compare_parts` function so that parts are sorted with their numbers in *descending* order.

19. Write a function that, when given a string as its argument, searches the following array of structures for a matching command name, then calls the function associated with that name.

```
struct {
  char *cmd_name;
  void (*cmd_pointer)(void);
} file_cmd[] =
 {{"new",        new_cmd},
  {"open",       open_cmd},
  {"close",      close_cmd},
  {"close all", close_all_cmd},
  {"save",       save_cmd},
  {"save as",    save_as_cmd},
  {"save all",   save_all_cmd},
  {"print",      print_cmd},
  {"exit",       exit_cmd}
};
```

Programming Projects

1. Modify the `inventory.c` program of Section 16.3 so that the `inventory` array is allocated dynamically and later reallocated when it fills up. Use `malloc` initially to allocate enough space for an array of 10 `part` structures. When the array has no more room for new parts, use `realloc` to double its size. Repeat the doubling step each time the array becomes full.

2. Modify the `inventory.c` program of Section 16.3 so that the `p` (print) command calls `qsort` to sort the `inventory` array before it prints the parts.

3. Modify the `inventory2.c` program of Section 17.5 by adding an `e` (erase) command that allows the user to remove a part from the database.

4. Modify the `justify` program of Section 15.3 by rewriting the `line.c` file so that it stores the current line in a linked list. Each node in the list will store a single word. The `line` array will be replaced by a variable that points to the node containing the first word. This variable will store a null pointer whenever the line is empty.

5. Write a program that sorts a series of words entered by the user:

```
Enter word: foo
Enter word: bar
Enter word: baz
Enter word: quux
Enter word:

In sorted order: bar baz foo quux
```

   Assume that each word is no more than 20 characters long. Stop reading when the user enters an empty word (i.e., presses Enter without entering a word). Store each word in a dynamically allocated string, using an array of pointers to keep track of the strings, as in the `remind2.c` program (Section 17.2). After all words have been read, sort the array (using any sorting technique) and then use a loop to print the words in sorted order. *Hint:* Use the `read_line` function to read each word, as in `remind2.c`.

6. Modify Programming Project 5 so that it uses `qsort` to sort the array of pointers.

7. (C99) Modify the `remind2.c` program of Section 17.2 so that each element of the `reminders` array is a pointer to a `vstring` structure (see Section 17.9) rather than a pointer to an ordinary string.

18 Declarations

Making something variable is easy.
Controlling duration of constancy is the trick.

Declarations play a central role in C programming. By declaring variables and functions, we furnish vital information that the compiler will need in order to check a program for potential errors and translate it into object code.

Previous chapters have provided examples of declarations without going into full details; this chapter fills in the gaps. It explores the sophisticated options that can be used in declarations and reveals that variable declarations and function declarations have quite a bit in common. It also provides a firm grounding in the important concepts of storage duration, scope, and linkage.

Section 18.1 examines the syntax of declarations in their most general form, a topic that we've avoided up to this point. The next four sections focus on the items that appear in declarations: storage classes (Section 18.2), type qualifiers (Section 18.3), declarators (Section 18.4), and initializers (Section 18.5). Section 18.6 discusses the `inline` keyword, which can appear in C99 function declarations.

18.1 Declaration Syntax

Declarations furnish information to the compiler about the meaning of identifiers. When we write

```
int i;
```

we're informing the compiler that, in the current scope, the name `i` represents a variable of type `int`. The declaration

```
float f(float);
```

tells the compiler that `f` is a function that returns a `float` value and has one argument, also of type `float`.

In general, a declaration has the following appearance:

declaration

declaration-specifiers declarators ;

Declaration specifiers describe the properties of the variables or functions being declared. ***Declarators*** give their names and may provide additional information about their properties.

Declaration specifiers fall into three categories:

- ***Storage classes.*** There are four storage classes: `auto`, `static`, `extern`, and `register`. At most one storage class may appear in a declaration; if present, it should come first.
- ***Type qualifiers.*** In C89, there are only two type qualifiers: `const` and `volatile`. C99 has a third type qualifier, `restrict`. A declaration may contain zero or more type qualifiers.

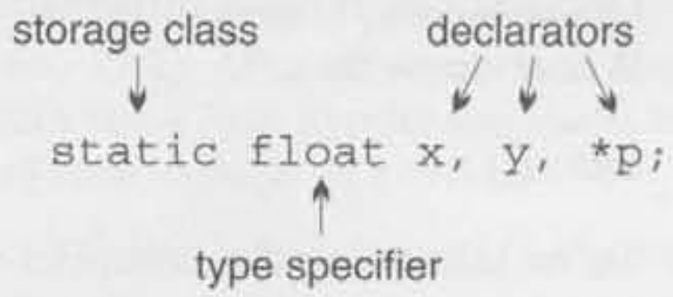
C99

- ***Type specifiers.*** The keywords `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, and `unsigned` are all type specifiers. These words may be combined as described in Chapter 7; the order in which they appear doesn't matter (`int unsigned long` is the same as `long unsigned int`). Type specifiers also include specifications of structures, unions, and enumerations (for example, `struct point { int x, y; }`, `struct { int x, y; }`, or `struct point`). Type names created using `typedef` are type specifiers as well.

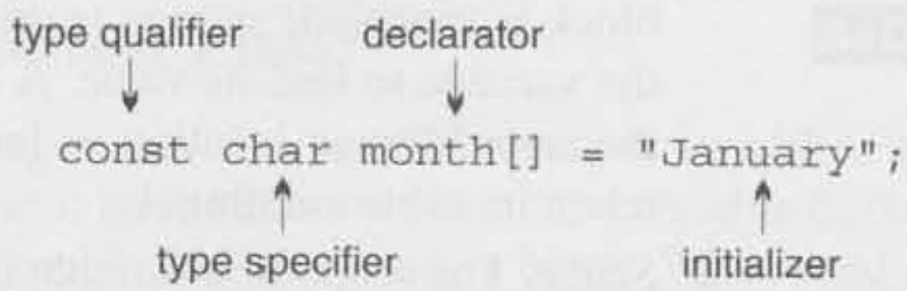
C99 (C99 has a fourth kind of declaration specifier, the ***function specifier,*** which is used only in function declarations. This category has just one member, the keyword `inline`.) Type qualifiers and type specifiers should follow the storage class, but there are no other restrictions on their order. As a matter of style, I'll put type qualifiers before type specifiers.

Declarators include identifiers (names of simple variables), identifiers followed by `[]` (array names), identifiers preceded by `*` (pointer names), and identifiers followed by `()` (function names). Declarators are separated by commas. A declarator that represents a variable may be followed by an initializer.

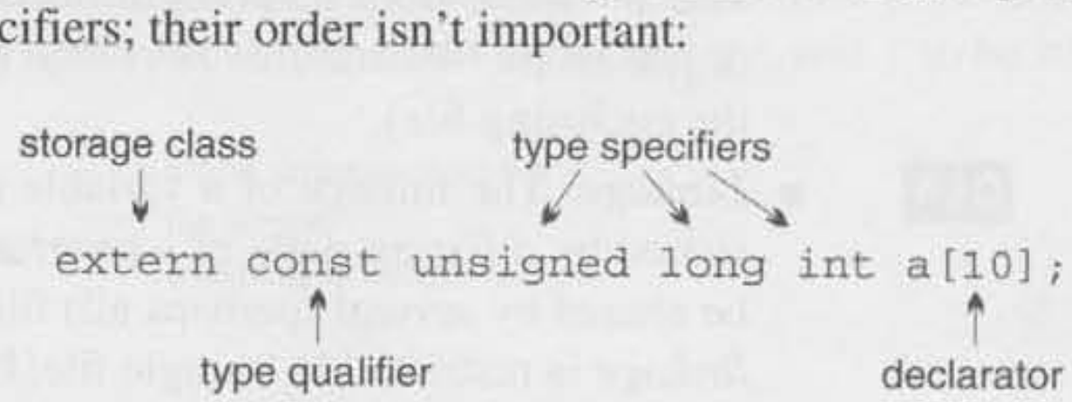
Let's look at a few examples that illustrate these rules. Here's a declaration with a storage class and three declarators:



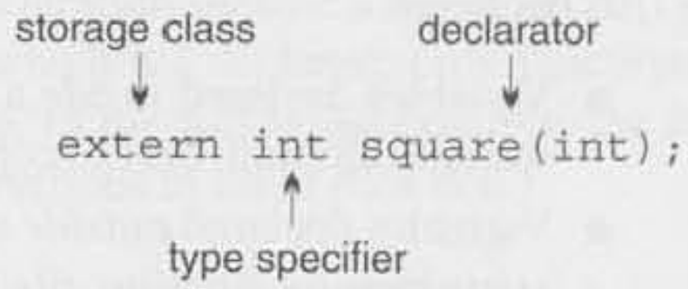
The following declaration has a type qualifier but no storage class. It also has an initializer:



The following declaration has both a storage class and a type qualifier. It also has three type specifiers; their order isn't important:



Function declarations, like variable declarations, may have a storage class, type qualifiers, and type specifiers. The following declaration has a storage class and a type specifier:



The next four sections cover storage classes, type qualifiers, declarators, and initializers in detail.

18.2 Storage Classes

Storage classes can be specified for variables and—to a lesser extent—functions and parameters. We'll concentrate on variables for now.

Recall from Section 10.3 that the term *block* refers to the body of a function (the part enclosed in braces) or a compound statement, possibly containing declarations. In C99, selection statements (`if` and `switch`) and iteration statements (`while`, `do`, and `for`)—along with the "inner" statements that they control—are considered to be blocks as well, although this is primarily a technicality.

C99

Q&A

Properties of Variables

Every variable in a C program has three properties:

- ***Storage duration.*** The storage duration of a variable determines when memory is set aside for the variable and when that memory is released. Storage for a variable with ***automatic storage duration*** is allocated when the surrounding

Q&A

block is executed; storage is deallocated when the block terminates, causing the variable to lose its value. A variable with ***static storage duration*** stays at the same storage location as long as the program is running, allowing it to retain its value indefinitely.

- ***Scope.*** The scope of a variable is the portion of the program text in which the variable can be referenced. A variable can have either ***block scope*** (the variable is visible from its point of declaration to the end of the enclosing block) or ***file scope*** (the variable is visible from its point of declaration to the end of the enclosing file).

Q&A

- ***Linkage.*** The linkage of a variable determines the extent to which it can be shared by different parts of a program. A variable with ***external linkage*** may be shared by several (perhaps all) files in a program. A variable with ***internal linkage*** is restricted to a single file, but may be shared by the functions in that file. (If a variable with the same name appears in another file, it's treated as a different variable.) A variable with ***no linkage*** belongs to a single function and can't be shared at all.

The default storage duration, scope, and linkage of a variable depend on where it's declared:

- Variables declared *inside* a block (including a function body) have *automatic* storage duration, *block* scope, and *no* linkage.
- Variables declared *outside* any block, at the outermost level of a program, have *static* storage duration, *file* scope, and *external* linkage.

The following example shows the default properties of the variables `i` and `j`:

```
int i;    /* static storage duration, file scope, external linkage */

void f(void)
{
  int j;  /* automatic storage duration, block scope, no linkage */
}
```

For many variables, the default storage duration, scope, and linkage are satisfactory. When they aren't, we can alter these properties by specifying an explicit storage class: `auto`, `static`, `extern`, or `register`.

The `auto` Storage Class

The `auto` storage class is legal only for variables that belong to a block. An `auto` variable has automatic storage duration (not surprisingly), block scope, and no linkage. The `auto` storage class is almost never specified explicitly, since it's the default for variables declared inside a block.

The `static` Storage Class

The `static` storage class can be used with all variables, regardless of where they're declared, but it has a different effect on a variable declared outside a block than it does on a variable declared inside a block. When used *outside* a block, the word `static` specifies that a variable has internal linkage. When used *inside* a block, `static` changes the variable's storage duration from automatic to static. The following figure shows the effect of declaring `i` and `j` to be `static`:

```
static int i;        static storage duration
                     file scope
                     internal linkage

void f(void)
{
  static int j;      static storage duration
                     block scope
                     no linkage
}
```

When used in a declaration outside a block, `static` essentially hides a variable within the file in which it's declared; only functions that appear in the same file can see the variable. In the following example, the functions `f1` and `f2` both have access to `i`, but functions in other files don't:

```
static int i;

void f1(void)
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

This use of `static` can help implement a technique known as information hiding.

information hiding ➤19.2

A `static` variable declared within a block resides at the same storage location throughout program execution. Unlike automatic variables, which lose their values each time the program leaves the enclosing block, a `static` variable will retain its value indefinitely. `static` variables have some interesting properties:

- A `static` variable in a block is initialized only once, prior to program execution. An `auto` variable is initialized every time it comes into existence (provided, of course, that it has an initializer).
- Each time a function is called recursively, it gets a new set of `auto` variables. If it has a `static` variable, on the other hand, that variable is shared by all calls of the function.

- Although a function shouldn't return a pointer to an `auto` variable, there's nothing wrong with it returning a pointer to a `static` variable.

Declaring one of its variables to be `static` allows a function to retain information between calls in a "hidden" area that the rest of the program can't access. More often, however, we'll use `static` to make programs more efficient. Consider the following function:

```
char digit_to_hex_char(int digit)
{
  const char hex_chars[16] = "0123456789ABCDEF";

  return hex_chars[digit];
}
```

Each time the `digit_to_hex_char` function is called, the characters `0123456789ABCDEF` will be copied into the `hex_chars` array to initialize it. Now, let's make the array `static`:

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] = "0123456789ABCDEF";

  return hex_chars[digit];
}
```

Since `static` variables are initialized only once, we've improved the speed of `digit_to_hex_char`.

The `extern` Storage Class

The `extern` storage class enables several source files to share the same variable. Section 15.2 covered the essentials of using `extern`, so I won't devote much space to it here. Recall that the declaration

```
extern int i;
```

informs the compiler that `i` is an `int` variable, but doesn't cause it to allocate memory for `i`. In C terminology, this declaration is not a *definition* of `i`; it merely informs the compiler that we need access to a variable that's defined elsewhere (perhaps later in the same file, or—more often—in another file). A variable can have many *declarations* in a program but should have only one *definition*.

There's one exception to the rule that an `extern` declaration of a variable isn't a definition. An `extern` declaration that initializes a variable serves as a definition of the variable. For example, the declaration

```
extern int i = 0;
```

is effectively the same as

```
int i = 0;
```

This rule prevents multiple `extern` declarations from initializing a variable in different ways.

A variable in an `extern` declaration always has static storage duration. The scope of the variable depends on the declaration's placement. If the declaration is inside a block, the variable has block scope; otherwise, it has file scope:

Q&A

```
                          static storage duration
extern int i;             file scope
                          ? linkage

void f(void)
{
                          static storage duration
  extern int j;           block scope
                          ? linkage
}
```

Determining the linkage of an `extern` variable is a bit harder. If the variable was declared `static` earlier in the file (outside of any function definition), then it has internal linkage. Otherwise (the normal case), the variable has external linkage.

The `register` Storage Class

Using the `register` storage class in the declaration of a variable asks the compiler to store the variable in a register instead of keeping it in main memory like other variables. (A ***register*** is a storage area located in a computer's CPU. Data stored in a register can be accessed and updated faster than data stored in ordinary memory.) Specifying the storage class of a variable to be `register` is a request, not a command. The compiler is free to store a `register` variable in memory if it chooses.

The `register` storage class is legal only for variables declared in a block. A `register` variable has the same storage duration, scope, and linkage as an `auto` variable. However, a `register` variable lacks one property that an `auto` variable has: since registers don't have addresses, it's illegal to use the `&` operator to take the address of a `register` variable. This restriction applies even if the compiler has elected to store the variable in memory.

`register` is best used for variables that are accessed and/or updated frequently. For example, the loop control variable in a `for` statement is a good candidate for `register` treatment:

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

`register` isn't nearly as popular among C programmers as it once was. Today's compilers are much more sophisticated than early C compilers; many can determine automatically which variables would benefit the most from being kept in registers. Still, using `register` provides useful information that can help the compiler optimize the performance of a program. In particular, the compiler knows that a `register` variable can't have its address taken, and therefore can't be modified through a pointer. In this respect, the `register` keyword is related to C99's `restrict` keyword.

The Storage Class of a Function

Function declarations (and definitions), like variable declarations, may include a storage class, but the only options are `extern` and `static`. The word `extern` at the beginning of a function declaration specifies that the function has external linkage, allowing it to be called from other files. `static` indicates internal linkage, limiting use of the function's name to the file in which it's defined. If no storage class is specified, the function is assumed to have external linkage.

Consider the following function declarations:

```
extern int f(int i);
static int g(int i);
int h(int i);
```

`f` has external linkage, `g` has internal linkage, and `h` (by default) has external linkage. Because it has internal linkage, `g` can't be called directly from outside the file in which it's defined. (Declaring `g` to be `static` doesn't completely prevent it from being called in another file; an indirect call via a function pointer is still possible.)

Declaring functions to be `extern` is like declaring variables to be `auto`—it serves no purpose. For that reason, I don't use `extern` in function declarations. Be aware, however, that some programmers use `extern` extensively, which certainly does no harm.

Declaring functions to be `static`, on the other hand, is quite useful. In fact, I recommend using `static` when declaring any function that isn't intended to be called from other files. The benefits of doing so include:

- ***Easier maintenance.*** Declaring a function `f` to be `static` guarantees that `f` isn't visible outside the file in which its definition appears. As a result, someone modifying the program later knows that changes to `f` won't affect functions in other files. (One exception: a function in another file that's passed a pointer to `f` might be affected by changes to `f`. Fortunately, that situation is easy to spot by examining the file in which `f` is defined, since the function that passes `f` must also be defined there.)
- ***Reduced "name space pollution."*** Since functions declared `static` have internal linkage, their names can be reused in other files. Although we proba-

bly wouldn't deliberately reuse a function name for some other purpose, it can be hard to avoid in large programs. An excessive number of names with external linkage can result in what C programmers call "name space pollution": names in different files accidentally conflicting with each other. Using `static` helps prevent this problem.

Function parameters have the same properties as `auto` variables: automatic storage duration, block scope, and no linkage. The only storage class that can be specified for parameters is `register`.

Summary

Now that we've covered the various storage classes, let's summarize what we know. The following program fragment shows all possible ways to include—or omit—storage classes in declarations of variables and parameters.

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```

Table 18.1 shows the properties of each variable and parameter in this example.

Table 18.1
Properties of Variables and Parameters

Name	*Storage Duration*	*Scope*	*Linkage*
a	static	file	external
b	static	file	†
c	static	file	internal
d	automatic	block	none
e	automatic	block	none
g	automatic	block	none
h	automatic	block	none
i	static	block	none
j	static	block	†
k	automatic	block	none

†The definitions of `b` and `j` aren't shown, so it's not possible to determine the linkage of these variables. In most cases, the variables will be defined in another file and will have external linkage.

Of the four storage classes, the most important are `static` and `extern`. `auto` has no effect, and modern compilers have made `register` less important.

18.3 Type Qualifiers

C99

restricted pointers ➤ 17.8

There are two type qualifiers: `const` and `volatile`. (C99 has a third type qualifier, `restrict`, which is used only with pointers.) Since the use of `volatile` is limited to low-level programming, I'll postpone discussing it until Section 20.3. `const` is used to declare objects that resemble variables but are "read-only": a program may access the value of a `const` object, but can't change it. For example, the declaration

```
const int n = 10;
```

creates a `const` object named `n` whose value is 10. The declaration

```
const int tax_brackets[] = {750, 2250, 3750, 5250, 7000};
```

creates a `const` array named `tax_brackets`.

Declaring an object to be `const` has several advantages:

- It's a form of documentation: it alerts anyone reading the program to the read-only nature of the object.
- The compiler can check that the program doesn't inadvertently attempt to change the value of the object.
- When programs are written for certain types of applications (embedded systems, in particular), the compiler can use the word `const` to identify data to be stored in ROM (read-only memory).

At first glance, it might appear that `const` serves the same role as the `#define` directive, which we've used in previous chapters to create names for constants. There are significant differences between `#define` and `const`, however:

- We can use `#define` to create a name for a numerical, character, or string constant. `const` can be used to create read-only objects of *any* type, including arrays, pointers, structures, and unions.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't. In particular, we can't use `#define` to create a constant with block scope.
- The value of a `const` object, unlike the value of a macro, can be viewed in a debugger.

Q&A

- Unlike macros, `const` objects can't be used in constant expressions. For example, we can't write

```
const int n = 10;
int a[n];               /*** WRONG ***/
```

C99

since array bounds must be constant expressions. (In C99, this example would

be legal if a has automatic storage duration—it would be treated as a variable-length array—but not if it has static storage duration.)

- It's legal to apply the address operator (&) to a `const` object, since it has an address. A macro doesn't have an address.

There are no absolute rules that dictate when to use `#define` and when to use `const`. I recommend using `#define` for constants that represent numbers or characters. That way, you'll be able to use the constants as array dimensions, in `switch` statements, and in other places where constant expressions are required.

18.4 Declarators

A declarator consists of an identifier (the name of the variable or function being declared), possibly preceded by the * symbol or followed by `[]` or `()`. By combining *, `[]`, and `()`, we can create declarators of mind-numbing complexity.

Before we look at the more complicated declarators, let's review the declarators that we've seen in previous chapters. In the simplest case, a declarator is just an identifier, like `i` in the following example:

```
int i;
```

Declarators may also contain the symbols *, `[]`, and `()`:

- A declarator that begins with * represents a pointer:

  ```
  int *p;
  ```

- A declarator that ends with `[]` represents an array:

  ```
  int a[10];
  ```

  The brackets may be left empty if the array is a parameter, if it has an initializer, or if its storage class is `extern`:

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

  Since `a` is defined elsewhere in the program, the compiler doesn't need to know its length here. (In the case of a multidimensional array, only the first set of brackets can be empty.) C99 provides two additional options for what goes between the brackets in the declaration of an array parameter. One option is the keyword `static`, followed by an expression that specifies the array's minimum length. The other is the * symbol, which can be used in a function prototype to indicate a variable-length array argument. Section 9.3 discusses both C99 features.

  C99

- A declarator that ends with `()` represents a function:

  ```
  int abs(int i);
  void swap(int *a, int *b);
  int find_largest(int a[], int n);
  ```

C allows parameter names to be omitted in a function declaration:

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

The parentheses can even be left empty:

```
int abs();
void swap();
int find_largest();
```

The declarations in the last group specify the return types of the `abs`, `swap`, and `find_largest` functions, but provide no information about their arguments. Leaving the parentheses empty isn't the same as putting the word `void` between them, which indicates that there are no arguments. The empty-parentheses style of function declaration has largely disappeared. It's inferior to the prototype style introduced in C89, since it doesn't allow the compiler to check whether function calls have the right arguments.

If all declarators were as simple as these, C programming would be a snap. Unfortunately, declarators in actual programs often combine the `*`, `[]`, and `()` notations. We've seen examples of such combinations already. We know that

```
int *ap[10];
```

declares an array of 10 pointers to integers. We know that

```
float *fp(float);
```

declares a function that has a `float` argument and returns a pointer to a `float`. And, in Section 17.7, we learned that

```
void (*pf)(int);
```

declares a pointer to a function with an `int` argument and a `void` return type.

Deciphering Complex Declarations

So far, we haven't had too much trouble understanding declarators. But what about declarators like the one in the following declaration?

```
int *(*x[10])(void);
```

This declarator combines `*`, `[]`, and `()`, so it's not obvious whether `x` is a pointer, an array, or a function.

Fortunately, there are two simple rules that will allow us to understand any declaration, no matter how convoluted:

- ***Always read declarators from the inside out.*** In other words, locate the identifier that's being declared, and start deciphering the declaration from there.

- ***When there's a choice, always favor*** `[]` ***and*** `()` ***over*** `*`. If `*` precedes the identifier and `[]` follows it, the identifier represents an array, not a pointer. Likewise, if `*` precedes the identifier and `()` follows it, the identifier represents a function, not a pointer. (Of course, we can always use parentheses to override the normal priority of `[]` and `()` over `*`.)

  Let's apply these rules to our simple examples first. In the declaration

```
int *ap[10];
```

  the identifier is `ap`. Since `*` precedes `ap` and `[]` follows it, we give preference to `[]`, so `ap` is an *array* of *pointers*. In the declaration

```
float *fp(float);
```

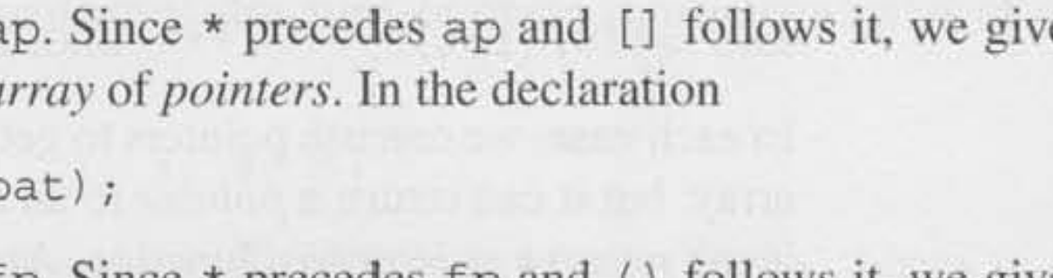
  the identifier is `fp`. Since `*` precedes `fp` and `()` follows it, we give preference to `()`, so `fp` is a *function* that returns a *pointer*.

  The declaration

```
void (*pf)(int);
```

  is a little trickier. Since `*pf` is enclosed in parentheses, `pf` must be a pointer. But `(*pf)` is followed by `(int)`, so `pf` must point to a function with an `int` argument. The word `void` represents the return type of this function.

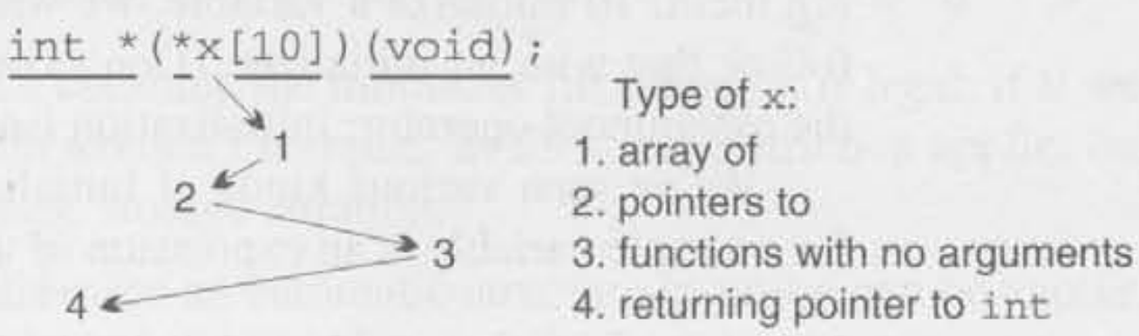
  As the last example shows, understanding a complex declarator often involves zigzagging from one side of the identifier to the other:



  Let's use this zigzagging technique to decipher the declaration given earlier:

```
int *(*x[10])(void);
```

  First, we locate the identifier being declared (`x`). We see that `x` is preceded by `*` and followed by `[]`; since `[]` have priority over `*`, we go right (`x` is an array). Next, we go left to find out the type of the elements in the array (pointers). Next, we go right to find out what kind of data the pointers point to (functions with no arguments). Finally, we go left to see what each function returns (a pointer to an `int`). Graphically, here's what the process looks like:



Mastering C declarations takes time and practice. The only good news is that there are certain things that can't be declared in C. Functions can't return arrays:

```
int f(int)[];        /*** WRONG ***/
```

Functions can't return functions:

```
int g(int)(int);     /*** WRONG ***/
```

Arrays of functions aren't possible, either:

```
int a[10](int);      /*** WRONG ***/
```

In each case, we can use pointers to get the desired effect. A function can't return an array, but it can return a *pointer* to an array. A function can't return a function, but it can return a *pointer* to a function. Arrays of functions aren't allowed, but an array may contain *pointers* to functions. (Section 17.7 has an example of such an array.)

Using Type Definitions to Simplify Declarations

Some programmers use type definitions to help simplify complex declarations. Consider the declaration of `x` that we examined earlier in this section:

```
int *(*x[10])(void);
```

To make `x`'s type easier to understand, we could use the following series of type definitions:

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

If we read these lines in reverse order, we see that `x` has type `Fcn_ptr_array`, a `Fcn_ptr_array` is an array of `Fcn_ptr` values, a `Fcn_ptr` is a pointer to type `Fcn`, and a `Fcn` is a function that has no arguments and returns a pointer to an `int` value.

18.5 Initializers

For convenience, C allows us to specify initial values for variables as we're declaring them. To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer. (Don't confuse the = symbol in a declaration with the assignment operator; initialization isn't the same as assignment.)

We've seen various kinds of initializers in previous chapters. The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5 / 2;    /* i is initially 2 */
```

If the types don't match, C converts the initializer using the same rules as for assignment:

conversion during assignment ➤7.4

```
int j = 5.5;      /* converted to 5 */
```

The initializer for a pointer variable must be a pointer expression of the same type as the variable or of type `void *`:

```
int *p = &i;
```

The initializer for an array, structure, or union is usually a series of values enclosed in braces:

```
int a[5] = {1, 2, 3, 4, 5};
```

C99

In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.

designated initializers ➤8.1, 16.1

To complete our coverage of declarations, let's take a look at some additional rules that govern initializers:

- An initializer for a variable with static storage duration must be constant:

  ```
  #define FIRST 1
  #define LAST 100
  static int i = LAST - FIRST + 1;
  ```

  Since `LAST` and `FIRST` are macros, the compiler can compute the initial value of `i` (100 – 1 + 1 = 100). If `LAST` and `FIRST` had been variables, the initializer would be illegal.

- If a variable has automatic storage duration, its initializer need not be constant:

  ```
  int f(int n)
  {
    int last = n - 1;
    …
  }
  ```

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions, never variables or function calls:

  ```
  #define N 2

  int powers[5] = {1, N, N * N, N * N * N, N * N * N * N};
  ```

  C99

  Since `N` is a constant, the initializer for `powers` is legal; if `N` were a variable, the program wouldn't compile. In C99, this restriction applies only if the variable has static storage duration.

- The initializer for an automatic structure or union can be another structure or union:

```
void g(struct part part1)
{
  struct part part2 = part1;
  ...
}
```

The initializer doesn't have to be a variable or parameter name, although it does need to be an expression of the proper type. For example, `part2`'s initializer could be `*p`, where `p` is of type `struct part *`, or `f(part1)`, where `f` is a function that returns a `part` structure.

Uninitialized Variables

In previous chapters, we've implied that uninitialized variables have undefined values. That's not always true; the initial value of a variable depends on its storage duration:

- Variables with *automatic* storage duration have no default initial value. The initial value of an automatic variable can't be predicted and may be different each time the variable comes into existence.
- Variables with *static* storage duration have the value zero by default. Unlike memory allocated by `calloc`, which is simply set to zero bits, a static variable is correctly initialized based on its type: integer variables are initialized to 0, floating variables are initialized to 0.0, and pointer variables contain a null pointer.

`calloc` function ➤ 17.3

As a matter of style, it's better to provide initializers for static variables rather than rely on the fact that they're guaranteed to be zero. If a program accesses a variable that hasn't been initialized explicitly, someone reading the program later can't easily determine whether the variable is assumed to be zero or whether it's initialized by an assignment somewhere in the program.

18.6 Inline Functions (C99)

C99 function declarations have an additional option that doesn't exist in C89: they may contain the keyword `inline`. This keyword is a new breed of declaration specifier, distinct from storage classes, type qualifiers, and type specifiers. To understand the effect of `inline`, we'll need to visualize the machine instructions that are generated by a C compiler to handle the process of calling a function and returning from a function.

At the machine level, several instructions may need to be executed to prepare for the call, the call itself requires jumping to the first instruction in the function, and there may be additional instructions executed by the function itself as it begins to execute. If the function has arguments, they'll need to be copied (because C passes its arguments by value). Returning from a function requires a similar

amount of effort on both the part of the function that was called and the one that called it. The cumulative work required to call a function and later return from it is often referred to as "overhead," since it's extra work above and beyond what the function is really supposed to accomplish. Although the overhead of a function call slows the program by only a tiny amount, it may add up in certain situations, such as when a function is called millions or billions of times, when an older, slower processor is in use (as might be the case in an embedded system), or when a program has to meet very strict deadlines (as in a real-time system).

parameterized macros ➤ 14.3

In C89, the only way to avoid the overhead of a function call is to use a parameterized macro. Parameterized macros have certain drawbacks, though. C99 offers a better solution to this problem: create an ***inline function.*** The word "inline" suggests an implementation strategy in which the compiler replaces each call of the function by the machine instructions for the function. This technique avoids the usual overhead of a function call, although it may cause a minor increase in the size of the compiled program.

Declaring a function to be `inline` doesn't actually force the compiler to "inline" the function, however. It merely suggests that the compiler should try to make calls of the function as fast as possible, perhaps by performing an inline expansion when the function is called. The compiler is free to ignore this suggestion. In this respect, `inline` is similar to the `register` and `restrict` keywords, which the compiler may use to improve the performance of a program but may also choose to ignore.

Inline Definitions

An inline function has the keyword `inline` as one of its declaration specifiers:

```
inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

Here's where things get a bit complicated. `average` has external linkage, so other source files may contain calls of `average`. However, the definition of `average` isn't considered to be an external definition by the compiler (it's an ***inline definition*** instead), so attempting to call `average` from another file will be considered an error.

There are two ways to avoid this error. One option is to add the word `static` to the function definition:

```
static inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

`average` now has internal linkage, so it can't be called from other files. Other files may contain their own definitions of `average`, which might be the same as this definition or might be different.

The other option is to provide an external definition for `average` so that calls are permitted from other files. One way to do this is to write the `average` function a second time (without using `inline`) and put the second definition in a different source file. Doing so is legal, but it's not a good idea to have two versions of the same function, because we can't guarantee that they'll remain consistent when the program is modified.

Here's a better approach. First, we'll put the inline definition of `average` in a header file (let's name it `average.h`):

```
#ifndef AVERAGE_H
#define AVERAGE_H

inline double average(double a, double b)
{
  return (a + b) / 2;
}

#endif
```

Next, we'll create a matching source file, `average.c`:

```
#include "average.h"

extern double average(double a, double b);
```

Now, any file that needs to call the `average` function may simply include `average.h`, which contains the inline definition of `average`. The `average.c` file contains a prototype for `average` that uses the `extern` keyword, which causes the definition of `average` included from `average.h` to be treated as an external definition in `average.c`.

The general rule in C99 is that if all top-level declarations of a function in a particular file include `inline` but not `extern`, then the definition of the function in that file is inline. If the function is used anywhere in the program (including the file that contains its inline definition), then an external definition of the function will need to be provided by some other file. When the function is called, the compiler may choose to perform an ordinary call (using the function's external definition) or perform inline expansion (using the function's inline definition). There's no way to tell which choice the compiler will make, so it's crucial that the two definitions be consistent. The technique that we just discussed (using the `average.h` and `average.c` files) guarantees that the definitions are the same.

Restrictions on Inline Functions

Since inline functions are implemented in a way that's quite different from ordinary functions, they're subject to different rules and restrictions. Variables with static storage duration are a particular problem for inline functions with external linkage. Consequently, C99 imposes the following restrictions on an inline function with external linkage (but not on one with internal linkage):

- The function may not define a modifiable `static` variable.
- The function may not contain references to variables with internal linkage.

Such a function is allowed to define a variable that is both `static` and `const`, but each inline definition of the function may create its own copy of the variable.

Using Inline Functions with GCC

Some compilers, including GCC, supported inline functions prior to the C99 standard. As a result, their rules for using inline functions may vary from the standard. In particular, the scheme described earlier (using the `average.h` and `average.c` files) may not work with these compilers. Version 4.3 of GCC (not available at the time this book was written) is expected to support inline functions in the way described in the C99 standard.

Functions that are specified to be both `static` and `inline` should work fine, regardless of the version of GCC. This strategy is legal in C99 as well, so it's the safest bet. A `static inline` function can be used within a single file or placed in a header file and included into any source file that needs to call the function.

There's another way to share an inline function among multiple files that works with older versions of GCC but conflicts with C99. This technique involves putting a definition of the function in a header file, specifying that the function is both `extern` and `inline`, then including the header file into any source file that contains a call of the function. A second copy of the definition—without the words `extern` and `inline`—is placed in one of the source files. (That way, if the compiler is unable to "inline" the function for any reason, it will still have a definition.)

A final note about GCC: Functions are "inlined" only when optimization is requested via the `-O` command-line option.

Q & A

***Q: C99 Why are selection statements and iteration statements (and their "inner" statements) considered to be blocks in C99? [p. 459]**

A: This rather surprising rule stems from a problem that can occur when compound literals are used in selection statements and iteration statements. The problem has to do with the storage duration of compound literals, so let's take a moment to discuss that issue first.

compound literals ➤9.3, 16.2

The C99 standard states that the object represented by a compound literal has static storage duration if the compound literal occurs outside the body of a function. Otherwise, it has automatic storage duration; as a result, the memory occupied by the object is deallocated at the end of the block in which the compound literal appears. Consider the following function, which returns a `point` structure created using a compound literal:

```
struct point create_point(int x, int y)
{
  return (struct point) {x, y};
}
```

This function works correctly, because the object created by the compound literal will be copied when the function returns. The original object will no longer exist, but the copy will remain. Now suppose that we change the function slightly:

```
struct point *create_point(int x, int y)
{
  return &(struct point) {x, y};
}
```

This version of `create_point` suffers from undefined behavior, because it returns a pointer to an object that has automatic storage duration and won't exist after the function returns.

Now let's return to the question we started with: Why are selection statements and iteration statements considered to be blocks? Consider the following example:

```
/* Example 1 - if statement without braces */

double *coefficients, value;

if (polynomial_selected == 1)
  coefficients = (double[3]) {1.5, -3.0, 6.0};
else
  coefficients = (double[3]) {4.5, 1.0, -3.5};
value = evaluate_polynomial(coefficients);
```

This program fragment apparently behaves in the desired fashion (but read on). `coefficients` will point to one of two objects created by compound literals, and this object will still exist at the time `evaluate_polynomial` is called. Now consider what happens if we put braces around the "inner" statements—the ones controlled by the `if` statement:

```
/* Example 2 - if statement with braces */

double *coefficients, value;

if (polynomial_selected == 1) {
  coefficients = (double[3]) {1.5, -3.0, 6.0};
} else {
  coefficients = (double[3]) {4.5, 1.0, -3.5};
}
value = evaluate_polynomial(coefficients);
```

Now we're in trouble. Each compound literal causes an object to be created, but that object exists only within the block formed by the braces that enclose the statement in which the literal appears. By the time `evaluate_polynomial` is called, `coefficients` points to an object that no longer exists. The result: undefined behavior.

The creators of C99 were unhappy with this state of affairs, because programmers were unlikely to expect that simply adding braces within an `if` statement would cause undefined behavior. To avoid the problem, they decided that the inner statements would always be considered blocks. As a result, Example 1 and Example 2 are equivalent, with both exhibiting undefined behavior.

A similar problem can arise when a compound literal is part of the controlling expression of a selection statement or iteration statement. For this reason, each entire selection statement and iteration statement is considered to be a block as well (as though an invisible set of braces surrounds the entire statement). So, for example, an `if` statement with an `else` clause consists of three blocks: each of the two inner statements is a block, as is the entire `if` statement.

Q: You said that storage for a variable with automatic storage duration is allocated when the surrounding block is executed. Is this true for C99's variable-length arrays? [p. 460]

C99

A: No. Storage for a variable-length array isn't allocated at the beginning of the surrounding block, because the length of the array isn't yet known. Instead, it's allocated when the declaration of the array is reached during the execution of the block. In this respect, variable-length arrays are different from all other automatic variables.

Q: What exactly is the difference between "scope" and "linkage"? [p. 460]

A: Scope is for the benefit of the compiler, while linkage is for the benefit of the linker. The compiler uses the scope of an identifier to determine whether or not it's legal to refer to the identifier at a given point in a file. When the compiler translates a source file into object code, it notes which names have external linkage, eventually storing these names in a table inside the object file. Thus, the linker has access to names with external linkage; names with internal linkage or no linkage are invisible to the linker.

Q: I don't understand how a name could have block scope but external linkage. Could you elaborate? [p. 463]

A: Certainly. Suppose that one source file defines a variable `i`:

```
int i;
```

Let's assume that the definition of `i` lies outside any function, so `i` has external linkage by default. In another file, there's a function `f` that needs to access `i`, so the body of `f` declares `i` as `extern`:

```
void f(void)
{
  extern int i;
  …
}
```

In the first file, `i` has file scope. Within `f`, however, `i` has block scope. If other functions besides `f` need access to `i`, they'll need to declare it separately. (Or we

can simply move the declaration of `i` outside `f` so that `i` has file scope.) What's confusing about this entire business is that each declaration or definition of `i` establishes a different scope; sometimes it's file scope, and sometimes it's block scope.

***Q: Why can't `const` objects be used in constant expressions? `const` means "constant," right? [p. 466]**

A: In C, `const` means "read-only," not "constant." Let's look at a few examples that illustrate why `const` objects can't be used in constant expressions.

To start with, a `const` object might only be constant during its *lifetime*, not throughout the execution of the program. Suppose that a `const` object is declared inside a function:

```
void f(int n)
{
  const int m = n / 2;
  …
}
```

When `f` is called, `m` will be initialized to the value of `n / 2`. The value of `m` will then remain constant until `f` returns. When `f` is called the next time, `m` will likely be given a different value. That's where the problem arises. Suppose that `m` appears in a `switch` statement:

```
void f(int n)
{
  const int m = n / 2;
  …
  switch (…) {
    …
    case m: …    /*** WRONG ***/
    …
  }
  …
}
```

The value of `m` won't be known until `f` is called, which violates C's rule that the values of case labels must be constant expressions.

Next, let's look at `const` objects declared outside blocks. These objects have external linkage and can be shared among files. If C allowed the use of `const` objects in constant expressions, we could easily find ourselves in the following situation:

```
extern const int n;
int a[n];    /*** WRONG ***/
```

`n` is probably defined in another file, making it impossible for the compiler to determine `a`'s length. (I'm assuming that `a` is an external variable, so it can't be a variable-length array.)

If that's not enough to convince you, consider this: If a `const` object is also declared to be `volatile`, its value may change at any time during execution. Here's an example from the C standard:

`volatile` type qualifier ➤20.3

```
extern const volatile int real_time_clock;
```

The `real_time_clock` variable may not be changed by the program (because it's declared `const`), yet its value may change via some other mechanism (because it's declared `volatile`).

Q: Why is the syntax of declarators so odd?

A: Declarations are intended to mimic use. A pointer declarator has the form `*p`, which matches the way the indirection operator will later be applied to `p`. An array declarator has the form `a[...]`, which matches the way the array will later be subscripted. A function declarator has the form `f(...)`, which matches the syntax of a function call. This reasoning extends to even the most complicated declarators. Consider the `file_cmd` array of Section 17.7, whose elements are pointers to functions. The declarator for `file_cmd` has the form

```
(*file_cmd[])(void)
```

and a call of one of the functions has the form

```
(*file_cmd[n])();
```

The parentheses, brackets, and * are in identical positions.

Exercises

Section 18.1

1. For each of the following declarations, identify the storage class, type qualifiers, type specifiers, declarators, and initializers.

   (a) `static char **lookup(int level);`
   (b) `volatile unsigned long io_flags;`
   (c) `extern char *file_name[MAX_FILES], path[];`
   (d) `static const char token_buf[] = "";`

Section 18.2

2. Answer each of the following questions with `auto`, `extern`, `register`, and/or `static`.

   (a) Which storage class is used primarily to indicate that a variable or function can be shared by several files?
   (b) Suppose that a variable `x` is to be shared by several functions in one file but hidden from functions in other files. Which storage class should `x` be declared to have?
   (c) Which storage classes can affect the storage duration of a variable?

3. List the storage duration (static or automatic), scope (block or file), and linkage (internal, external, or none) of each variable and parameter in the following file:

```
extern float a;

void f(register double b)
{
  static int c;
  auto char d;
}
```

W 4. Let `f` be the following function. What will be the value of `f(10)` if `f` has never been called before? What will be the value of `f(10)` if `f` has been called five times previously?

```
int f(int i)
{
  static int j = 0;
  return i * j++;
}
```

5. State whether each of the following statements is true or false. Justify each answer.
   (a) Every variable with static storage duration has file scope.
   (b) Every variable declared inside a function has no linkage.
   (c) Every variable with internal linkage has static storage duration.
   (d) Every parameter has block scope.

6. The following function is supposed to print an error message. Each message is preceded by an integer, indicating the number of times the function has been called. Unfortunately, the function always displays 1 as the number of the error message. Locate the error and show how to fix it without making any changes outside the function.

```
void print_error(const char *message)
{
  int n = 1;
  printf("Error %d: %s\n", n++, message);
}
```

Section 18.3

7. Suppose that we declare `x` to be a `const` object. Which one of the following statements about `x` is *false*?
   (a) If `x` is of type `int`, it can be used as the value of a case label in a `switch` statement.
   (b) The compiler will check that no assignment is made to `x`.
   (c) `x` is subject to the same scope rules as variables.
   (d) `x` can be of any type.

Section 18.4

W 8. Write a complete description of the type of `x` as specified by each of the following declarations.
   (a) `char (*x[10])(int);`
   (b) `int (*x(int))[5];`
   (c) `float *(*x(void))(int);`
   (d) `void (*x(int, void (*y)(int)))(int);`

9. Use a series of type definitions to simplify each of the declarations in Exercise 8.

W 10. Write declarations for the following variables and functions:
   (a) `p` is a pointer to a function with a character pointer argument that returns a character pointer.

(b) `f` is a function with two arguments: `p`, a pointer to a structure with tag `t`, and `n`, a long integer. `f` returns a pointer to a function that has no arguments and returns nothing.
(c) `a` is an array of four pointers to functions that have no arguments and return nothing. The elements of `a` initially point to functions named `insert`, `search`, `update`, and `print`.
(d) `b` is an array of 10 pointers to functions with two `int` arguments that return structures with tag `t`.

11. In Section 18.4, we saw that the following declarations are illegal:

```
int f(int)[];       /* functions can't return arrays     */
int g(int)(int);    /* functions can't return functions  */
int a[10](int);     /* array elements can't be functions */
```

We can, however, achieve similar effects by using pointers: a function can return a *pointer* to the first element in an array, a function can return a *pointer* to a function, and the elements of an array can be *pointers* to functions. Revise each of these declarations accordingly.

*12. (a) Write a complete description of the type of the function `f`, assuming that it's declared as follows:

```
int (*f(float (*)(long), char *))(double);
```

(b) Give an example showing how `f` would be called.

Section 18.5

W 13. Which of the following declarations are legal? (Assume that `PI` is a macro that represents 3.14159.)

```
(a) char c = 65;
(b) static int i = 5, j = i * i;
(c) double d = 2 * PI;
(d) double angles[] = {0, PI / 2, PI, 3 * PI / 2};
```

14. Which kind of variables cannot be initialized?
(a) Array variables
(b) Enumeration variables
(c) Structure variables
(d) Union variables
(e) None of the above

W 15. Which property of a variable determines whether or not it has a default initial value?
(a) Storage duration
(b) Scope
(c) Linkage
(d) Type

19 Program Design

Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.

It's obvious that real-world programs are larger than the examples in this book, but you may not realize just how much larger. Faster CPUs and larger main memories have made it possible to write programs that would have been impractical just a few years ago. The popularity of graphical user interfaces has added greatly to the average length of a program. Most full-featured programs today are at least 100,000 lines long. Million-line programs are commonplace, and it's not unheard-of for a program to have 10 million lines or more.

Q&A

Although C wasn't designed for writing large programs, many large programs have in fact been written in C. It's tricky, and it requires a great deal of care, but it can be done. In this chapter, I'll discuss techniques that have proved to be helpful for writing large programs and show which C features (the `static` storage class, for example) are especially useful.

Writing large programs (often called "programming-in-the-large") is quite different from writing small ones—it's like the difference between writing a term paper (10 pages double-spaced, of course) and a 1000-page book. A large program requires more attention to style, since many people will be working on it. It requires careful documentation. It requires planning for maintenance, since it will likely be modified many times.

Above all, a large program requires careful design and much more planning than a small program. As Alan Kay, the designer of the SmallTalk programming language, puts it, "You can build a doghouse out of anything." A doghouse can be built without any particular design, using whatever materials are at hand. A house for humans, on the other hand, is too complex to just throw together.

Chapter 15 discussed writing large programs in C, but it concentrated on language details. In this chapter, we'll revisit the topic, this time focusing on techniques for good program design. A complete discussion of program design issues is obviously beyond the scope of this book. However, I'll try to cover—briefly—

some important concepts in program design and show how to use them to create C programs that are readable and maintainable.

Section 19.1 discusses how to view a C program as a collection of modules that provide services to each other. We'll then see how the concepts of information hiding (Section 19.2) and abstract data types (Section 19.3) can improve modules. By focusing on a single example (a stack data type), Section 19.4 illustrates how an abstract data type can be defined and implemented in C. Section 19.5 describes some limitations of C for defining abstract data types and shows how to work around them.

19.1 Modules

When designing a C program (or a program in any other language, for that matter), it's often useful to view it as a number of independent ***modules.*** A module is a collection of services, some of which are made available to other parts of the program (the ***clients***). Each module has an ***interface*** that describes the available services. The details of the module—including the source code for the services themselves—are stored in the module's ***implementation***.

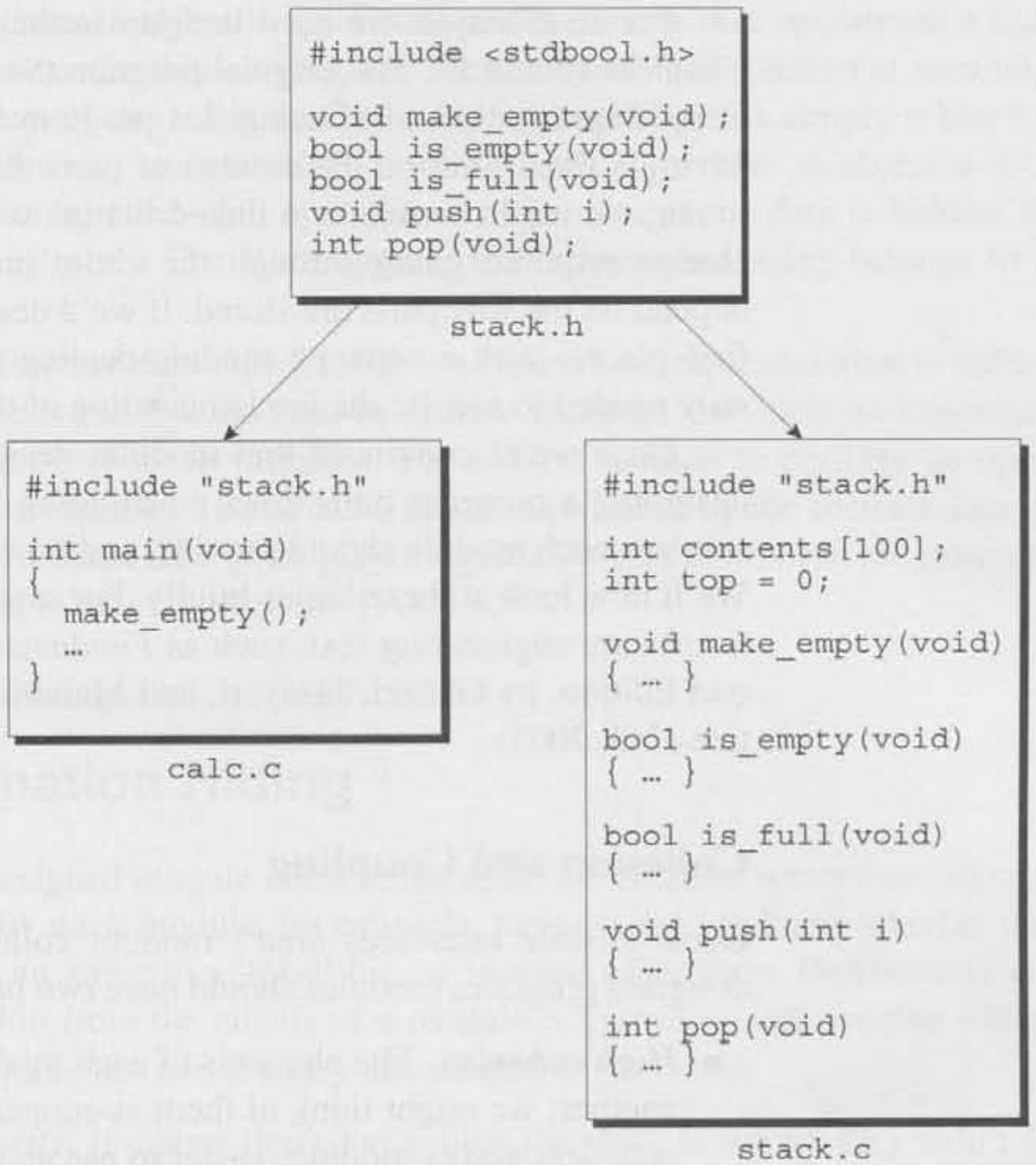
In the context of C, "services" are functions. The interface of a module is a header file containing prototypes for the functions that will be made available to clients (source files). The implementation of a module is a source file that contains definitions of the module's functions.

To illustrate this terminology, let's look at the calculator program that was sketched in Sections 15.1 and 15.2. This program consists of the file `calc.c`, which contains the `main` function, and a stack module, which is stored in the files `stack.h` and `stack.c` (see the figure at the top of the next page). `calc.c` is a *client* of the stack module. `stack.h` is the *interface* of the stack module; it supplies everything the client needs to know about the module. `stack.c` is the *implementation* of the module; it contains definitions of the stack functions as well as declarations of the variables that make up the stack.

The C library is itself a collection of modules. Each header in the library serves as the interface to a module. `<stdio.h>`, for example, is the interface to a module containing I/O functions, while `<string.h>` is the interface to a module containing string-handling functions.

Dividing a program into modules has several advantages:

- ***Abstraction.*** If modules are properly designed, we can treat them as ***abstractions***; we know what they do, but we don't worry about the details of how they do it. Thanks to abstraction, it's not necessary to understand how the entire program works in order to make changes to one part of it. What's more, abstraction makes it easier for several members of a team to work on the same program. Once the interfaces for the modules have been agreed upon, the responsibility for implementing each module can be delegated to a partic-



ular person. Team members can then work largely independently of one another.

- ***Reusability.*** Any module that provides services is potentially reusable in other programs. Our stack module, for example, is reusable. Since it's often hard to anticipate the future uses of a module, it's a good idea to design modules for reusability.
- ***Maintainability.*** A small bug will usually affect only a single module implementation, making the bug easier to locate and fix. Once the bug has been fixed, rebuilding the program requires only a recompilation of the module implementation (followed by linking the entire program). On a larger scale, we could replace an entire module implementation, perhaps to improve performance or when transporting the program to a different platform.

Although all these advantages are important, maintainability is the most critical. Most real-world programs are in service over a period of years, during which bugs are discovered, enhancements are made, and modifications are made to meet changing requirements. Designing a program in a modular fashion makes maintenance much easier. Maintaining a program should be like maintaining a car—fixing a flat tire shouldn't require overhauling the engine.

For an example, we need look no further than the `inventory` program of Chapters 16 and 17. The original program (Section 16.3) stored part records in an array. Suppose that, after using this program for a while, the customer objects to having a fixed limit on the number of parts that can be stored. To satisfy the customer, we might switch to a linked list (as we did in Section 17.5). Making this change required going through the entire program, looking for all places that depend on the way parts are stored. If we'd designed the program differently in the first place—with a separate module dealing with part storage—we would have only needed to rewrite the implementation of that module, not the entire program.

Once we're convinced that modular design is the way to go, the process of designing a program boils down to deciding what modules it should have, what services each module should provide, and how the modules should be interrelated. We'll now look at these issues briefly. For more information about design, consult a software engineering text, such as *Fundamentals of Software Engineering,* Second Edition, by Ghezzi, Jazayeri, and Mandrioli (Upper Saddle River, N.J.: Prentice-Hall, 2003).

Cohesion and Coupling

Good module interfaces aren't random collections of declarations. In a well-designed program, modules should have two properties:

- ***High cohesion.*** The elements of each module should be closely related to one another; we might think of them as cooperating toward a common goal. High cohesion makes modules easier to use and makes the entire program easier to understand.
- ***Low coupling.*** Modules should be as independent of each other as possible. Low coupling makes it easier to modify the program and reuse modules.

Does the calculator program have these properties? The stack module is clearly cohesive: its functions represent operations on a stack. There's little coupling in the program. The `calc.c` file depends on `stack.h` (and `stack.c` depends on `stack.h`, of course), but there are no other apparent dependencies.

Types of Modules

Because of the need for high cohesion and low coupling, modules tend to fall into certain typical categories:

- A ***data pool*** is a collection of related variables and/or constants. In C, a module of this type is often just a header file. From a design standpoint, putting variables in header files isn't usually a good idea, but collecting related constants in a header file can often be useful. In the C library, `<float.h>` and `<limits.h>` are both data pools.

`<float.h>` header ➤23.1
`<limits.h>` header ➤23.2

- A ***library*** is a collection of related functions. The `<string.h>` header, for example, is the interface to a library of string-handling functions.

- An ***abstract object*** is a collection of functions that operate on a hidden data structure. (In this chapter, the term "object" has a different meaning than in the rest of the book. In C terminology, an object is simply a block of memory that can store a value. In this chapter, however, an object is a collection of data bundled with operations on the data. If the data is hidden, the object is "abstract.") The stack module we've been discussing belongs to this category.
- An ***abstract data type (ADT)*** is a type whose representation is hidden. Client modules can use the type to declare variables, but have no knowledge of the structure of those variables. For a client module to perform an operation on such a variable, it must call a function provided by the abstract data type module. Abstract data types play a significant role in modern programming; we'll return to them in Sections 19.3–19.5.

19.2 Information Hiding

A well-designed module often keeps some information secret from its clients. Clients of our stack module, for example, have no need to know whether the stack is stored in an array, in a linked list, or in some other form. Deliberately concealing information from the clients of a module is known as ***information hiding***. Information hiding has two primary advantages:

- ***Security.*** If clients don't know how the stack is stored, they won't be able to corrupt it by tampering with its internal workings. To perform operations on the stack, they'll have to call functions that are provided by the module itself—functions that we've written and tested.
- ***Flexibility.*** Making changes—no matter how large—to a module's internal workings won't be difficult. For example, we could implement the stack as an array at first, then later switch to a linked list or other representation. We'll have to rewrite the implementation of the module, of course, but—if the module was designed properly—we won't have to alter the module's interface.

In C, the major tool for enforcing information hiding is the `static` storage class. Declaring a variable with file scope to be `static` gives it internal linkage, thus preventing it from being accessed from other files, including clients of the module. (Declaring a function to be `static` is also useful—the function can be directly called only by other functions in the same file.)

`static` storage class ➤ 18.2

A Stack Module

To see the benefits of information hiding, let's look at two implementations of a stack module, one using an array and the other a linked list. The module's header file will have the following appearance:

stack.h

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>   /* C99 only */

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

#endif
```

I've included C99's `<stdbool.h>` header so that the `is_empty` and `is_full` functions can return a `bool` result rather than an `int` value.

Let's first use an array to implement the stack:

stack1.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  top = 0;
}

bool is_empty(void)
{
  return top == 0;
}

bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    terminate("Error in push: stack is full.");
  contents[top++] = i;
}
```

```
int pop(void)
{
  if (is_empty())
    terminate("Error in pop: stack is empty.");
  return contents[--top];
}
```

The variables that make up the stack (`contents` and `top`) are both declared `static`, since there's no reason for the rest of the program to access them directly. The `terminate` function is also declared `static`. This function isn't part of the module's interface; instead, it's designed for use solely within the implementation of the module.

As a matter of style, some programmers use macros to indicate which functions and variables are "public" (accessible elsewhere in the program) and which are "private" (limited to a single file):

```
#define PUBLIC  /* empty */
#define PRIVATE static
```

The reason for writing `PRIVATE` instead of `static` is that the latter has more than one use in C; `PRIVATE` makes it clear that we're using it to enforce information hiding. Here's what the stack implementation would look like if we were to use `PUBLIC` and `PRIVATE`:

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PRIVATE void terminate(const char *message) { … }

PUBLIC void make_empty(void) { … }

PUBLIC bool is_empty(void) { … }

PUBLIC bool is_full(void) { … }

PUBLIC void push(int i) { … }

PUBLIC int pop(void) { … }
```

Now we'll switch to a linked-list implementation of the stack module:

stack2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
  int data;
  struct node *next;
};

static struct node *top = NULL;
```

```
static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  while (!is_empty())
    pop();
}

bool is_empty(void)
{
  return top == NULL;
}

bool is_full(void)
{
  return false;
}

void push(int i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = top;
  top = new_node;
}

int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty())
    terminate("Error in pop: stack is empty.");

  old_top = top;
  i = top->data;
  top = top->next;
  free(old_top);
  return i;
}
```

Note that the `is_full` function returns `false` every time it's called. A linked list has no limit on its size, so the stack will never be full. It's possible (but not likely) that the program might run out of memory, which will cause the `push` function to fail, but there's no easy way to test for that condition in advance.

Our stack example shows clearly the advantage of information hiding: it

doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module. Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

19.3 Abstract Data Types

A module that serves as an abstract object, like the stack module in the previous section, has a serious disadvantage: there's no way to have multiple instances of the object (more than one stack, in this case). To accomplish this, we'll need to go a step further and create a new *type*.

Once we've defined a `Stack` type, we'll be able to have as many stacks as we want. The following fragment illustrates how we could have two stacks in the same program:

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
  printf("%d\n", pop(&s1));    /* prints "1" */
```

We're not really sure what `s1` and `s2` are (structures? pointers?), but it doesn't matter. To clients, `s1` and `s2` are *abstractions* that respond to certain operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

Let's convert our `stack.h` header so that it provides a `Stack` type, where `Stack` is a structure. Doing so will require adding a `Stack` (or `Stack *`) parameter to each function. The header will now look like this (changes to `stack.h` are in **bold;** unchanged portions of the header aren't shown):

```
#define STACK_SIZE 100

typedef struct {
  int contents[STACK_SIZE];
  int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

The stack parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack. The parameter to `is_empty` and `is_full` doesn't need to be a pointer, but I've made it one anyway. Passing these functions a `Stack` *pointer* instead of a `Stack` *value* is more efficient, since the latter would result in a structure being copied.

Encapsulation

Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is. Nothing prevents clients from using a `Stack` variable as a structure:

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

Providing access to the `top` and `contents` members allows clients to corrupt the stack. Worse still, we won't be able to change the way stacks are stored without having to assess the effect of the change on clients.

What we need is a way to prevent clients from knowing how the `Stack` type is represented. C has only limited support for ***encapsulating*** types in this way. Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.

Incomplete Types

The only tool that C gives us for encapsulation is the ***incomplete type.*** (Incomplete types were mentioned briefly in Section 17.9 and in the Q&A section at the end of Chapter 17.) The C standard describes incomplete types as "types that describe objects but lack information needed to determine their sizes." For example, the declaration

Q&A

```
struct t;   /* incomplete declaration of t */
```

tells the compiler that `t` is a structure tag but doesn't describe the members of the structure. As a result, the compiler doesn't have enough information to determine the size of such a structure. The intent is that an incomplete type will be completed elsewhere in the program.

As long as a type remains incomplete, its uses are limited. Since the compiler doesn't know the size of an incomplete type, it can't be used to declare a variable:

Q&A

```
struct t s;   /*** WRONG ***/
```

However, it's perfectly legal to define a pointer type that references an incomplete type:

```
typedef struct t *T;
```

This type definition states that a variable of type `T` is a pointer to a structure with tag `t`. We can now declare variables of type `T`, pass them as arguments to functions, and perform other operations that are legal for pointers. (The size of a pointer doesn't depend on what it points to, which explains why C allows this behavior.) What we can't do, though, is apply the `->` operator to one of these variables, since the compiler knows nothing about the members of a `t` structure.

19.4 A Stack Abstract Data Type

To illustrate how abstract data types can be encapsulated using incomplete types, we'll develop a stack ADT based on the stack module described in Section 19.2. In the process, we'll explore three different ways to implement the stack.

Defining the Interface for the Stack ADT

First, we'll need a header file that defines our stack ADT type and gives prototypes for the functions that represent stack operations. Let's name this file `stackADT.h`. The `Stack` type will be a pointer to a `stack_type` structure that stores the actual contents of the stack. This structure is an incomplete type that will be completed in the file that implements the stack. The members of this structure will depend on how the stack is implemented. Here's what the `stackADT.h` file will look like:

stackADT.h (version 1)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Clients that include `stackADT.h` will be able to declare variables of type `Stack`, each of which is capable of pointing to a `stack_type` structure. Clients can then call the functions declared in `stackADT.h` to perform operations on stack variables. However, clients can't access the members of the `stack_type` structure, since that structure will be defined in a separate file.

Note that each function has a `Stack` parameter or returns a `Stack` value. The stack functions in Section 19.3 had parameters of type `Stack *`. The reason for the difference is that a `Stack` variable is now a pointer; it points to a `stack_type` structure that stores the contents of the stack. If a function needs to modify the stack, it changes the structure itself, not the pointer to the structure.

Also note the presence of the `create` and `destroy` functions. A module

generally doesn't need these functions, but an ADT does. `create` will dynamically allocate memory for a stack (including the memory required for a `stack_type` structure), as well as initializing the stack to its "empty" state. `destroy` will release the stack's dynamically allocated memory.

The following client file can be used to test the stack ADT. It creates two stacks and performs a variety of operations on them.

stackclient.c

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

int main(void)
{
  Stack s1, s2;
  int n;

  s1 = create();
  s2 = create();

  push(s1, 1);
  push(s1, 2);

  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);
  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);

  destroy(s1);

  while (!is_empty(s2))
    printf("Popped %d from s2\n", pop(s2));

  push(s2, 3);
  make_empty(s2);
  if (is_empty(s2))
    printf("s2 is empty\n");
  else
    printf("s2 is not empty\n");

  destroy(s2);

  return 0;
}
```

If the stack ADT is implemented correctly, the program should produce the following output:

```
Popped 2 from s1
Popped 1 from s1
Popped 1 from s2
Popped 2 from s2
s2 is empty
```

Implementing the Stack ADT Using a Fixed-Length Array

There are several ways to implement the stack ADT. Our first approach is the simplest. We'll have the `stackADT.c` file define the `stack_type` structure so that it contains a fixed-length array (to hold the contents of the stack) along with an integer that keeps track of the top of the stack:

```
struct stack_type {
  int contents[STACK_SIZE];
  int top;
};
```

Here's what `stackADT.c` will look like:

stackADT.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define STACK_SIZE 100

struct stack_type {
  int contents[STACK_SIZE];
  int top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = 0;
  return s;
}

void destroy(Stack s)
{
  free(s);
}

void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}
```

```
bool is_full(Stack s)
{
  return s->top == STACK_SIZE;
}

void push(Stack s, int i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

int pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

The most striking thing about the functions in this file is that they use the `->` operator, not the `.` operator, to access the `contents` and `top` members of the `stack_type` structure. The `s` parameter is a pointer to a `stack_type` structure, not a structure itself, so using the `.` operator would be illegal.

Changing the Item Type in the Stack ADT

Now that we have a working version of the stack ADT, let's try to improve it. First, note that items in the stack must be integers. That's too restrictive; in fact, the item type doesn't really matter. The stack items could just as easily be other basic types (`float`, `double`, `long`, etc.) or even structures, unions, or pointers, for that matter.

To make the stack ADT easier to modify for different item types, let's add a type definition to the `stackADT.h` header. It will define a type named `Item`, representing the type of data to be stored on the stack.

stackADT.h (version 2)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
```

```
void push(Stack s, Item i);
Item pop(Stack s);

#endif
```

The changes to the file are shown in **bold.** Besides the addition of the `Item` type, the `push` and `pop` functions have been modified. `push` now has a parameter of type `Item`, and `pop` returns a value of type `Item`. We'll use this version of `stackADT.h` from now on; it replaces the earlier version.

The `stackADT.c` file will need to be modified to match the new `stackADT.h`. The changes are minimal, however. The `stack_type` structure will now contain an array whose elements have type `Item` instead of `int`:

```
struct stack_type {
  Item contents[STACK_SIZE];
  int top;
};
```

The only other changes are to `push` (the second parameter now has type `Item`) and `pop` (which returns a value of type `Item`). The bodies of `push` and `pop` are unchanged.

The `stackclient.c` file can be used to test the new `stackADT.h` and `stackADT.c` to verify that the `Stack` type still works (it does!). Now we can change the item type any time we want by simply modifying the definition of the `Item` type in `stackADT.h`. (Although we won't have to change the `stackADT.c` file, we'll still need to recompile it.)

Implementing the Stack ADT Using a Dynamic Array

Another problem with the stack ADT as it currently stands is that each stack has a fixed maximum size, which is currently set at 100 items. This limit can be increased to any number we wish, of course, but all stacks created using the `Stack` type will have the same limit. There's no way to have stacks with different capacities or to set the stack size as the program is running.

There are two solutions to this problem. One is to implement the stack as a linked list, in which case there's no fixed limit on its size. We'll investigate this solution in a moment. First, though, let's try the other approach, which involves storing stack items in a dynamically allocated array.

dynamically allocated arrays ➤17.3

The crux of the latter approach is to modify the `stack_type` structure so that the `contents` member is a *pointer* to the array in which the items are stored, not the array itself:

```
struct stack_type {
  Item *contents;
  int top;
  int size;
};
```

I've also added a new member, `size`, that stores the stack's maximum size (the length of the array that `contents` points to). We'll use this member to check for the "stack full" condition.

The `create` function will now have a parameter that specifies the desired maximum stack size:

```
Stack create(int size);
```

When `create` is called, it will create a `stack_type` structure plus an array of length `size`. The `contents` member of the structure will point to this array.

The `stackADT.h` file will be the same as before, except that we'll need to add a `size` parameter to the `create` function. (Let's name the new version `stackADT2.h`.) The `stackADT.c` file will need more extensive modification, however. The new version appears below, with changes shown in **bold.**

stackADT2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT2.h"

struct stack_type {
  Item *contents;
  int top;
  int size;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->contents = malloc(size * sizeof(Item));
  if (s->contents == NULL) {
    free(s);
    terminate("Error in create: stack could not be created.");
  }
  s->top = 0;
  s->size = size;
  return s;
}

void destroy(Stack s)
{
  free(s->contents);
  free(s);
}
```

```
void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}

bool is_full(Stack s)
{
  return s->top == s->size;
}

void push(Stack s, Item i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

Item pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

The `create` function now calls `malloc` twice: once to allocate a `stack_type` structure and once to allocate the array that will contain the stack items. Either call of `malloc` could fail, causing `terminate` to be called. The `destroy` function must call `free` twice to release all the memory allocated by `create`.

The `stackclient.c` file can again be used to test the stack ADT. The calls of `create` will need to be changed, however, since `create` now requires an argument. For example, we could replace the statements

```
s1 = create();
s2 = create();
```

with the following statements:

```
s1 = create(100);
s2 = create(200);
```

Implementing the Stack ADT Using a Linked List

Implementing the stack ADT using a dynamically allocated array gives us more flexibility than using a fixed-size array. However, the client is still required to specify a maximum size for a stack at the time it's created. If we use a linked-list implementation instead, there won't be any preset limit on the size of a stack.

Our implementation will be similar to the one in the `stack2.c` file of Section 19.2. The linked list will consist of nodes, represented by the following structure:

```
struct node {
  Item data;
  struct node *next;
};
```

The type of the `data` member is now `Item` rather than `int`, but the structure is otherwise the same as before.

The `stack_type` structure will contain a pointer to the first node in the list:

```
struct stack_type {
  struct node *top;
};
```

At first glance, the `stack_type` structure seems superfluous; we could just define `Stack` to be `struct node *` and let a `Stack` value be a pointer to the first node in the list. However, we still need the `stack_type` structure so that the interface to the stack remains unchanged. (If we did away with it, any function that modified the stack would need a `Stack *` parameter instead of a `Stack` parameter.) Moreover, having the `stack_type` structure will make it easier to change the implementation in the future, should we decide to store additional information. For example, if we later decide that the `stack_type` structure should contain a count of how many items are currently stored in the stack, we can easily add a member to the `stack_type` structure to store this information.

We won't need to make any changes to the `stackADT.h` header. (We'll use this header file, not `stackADT2.h`.) We can also use the original `stackclient.c` file for testing. All the changes will be in the `stackADT.c` file. Here's the new version:

stackADT3.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}

void destroy(Stack s)
{
  make_empty(s);
  free(s);
}

void make_empty(Stack s)
{
  while (!is_empty(s))
    pop(s);
}

bool is_empty(Stack s)
{
  return s->top == NULL;
}

bool is_full(Stack s)
{
  return false;
}

void push(Stack s, Item i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = s->top;
  s->top = new_node;
}

Item pop(Stack s)
{
  struct node *old_top;
  Item i;

  if (is_empty(s))
    terminate("Error in pop: stack is empty.");

  old_top = s->top;
  i = old_top->data;
  s->top = old_top->next;
  free(old_top);
  return i;
}
```

Note that the `destroy` function calls `make_empty` (to release the memory occupied by the nodes in the linked list) before it calls `free` (to release the memory for the `stack_type` structure).

19.5 Design Issues for Abstract Data Types

Section 19.4 described a stack ADT and showed several ways to implement it. Unfortunately, this ADT suffers from several problems that prevent it from being industrial-strength. Let's look at each of these problems and discuss possible solutions.

Naming Conventions

The stack ADT functions currently have short, easy-to-understand names: `create`, `destroy`, `make_empty`, `is_empty`, `is_full`, `push`, and `pop`. If we have more than one ADT in a program, name clashes are likely, with functions in two modules having the same name. (Each ADT will need its own `create` function, for example.) Therefore, we'll probably need to use function names that incorporate the name of the ADT itself, such as `stack_create` instead of `create`.

Error Handling

The stack ADT deals with errors by displaying an error message and terminating the program. That's not a bad thing to do. The programmer can avoid popping an empty stack or pushing data onto a full stack by being careful to call `is_empty` prior to each call of `pop` and `is_full` prior to each call of `push`, so in theory there's no reason for a call of `push` or `pop` to fail. (In the linked-list implementation, however, calling `is_full` isn't foolproof; a subsequent call of `push` can still fail.) Nevertheless, we might want to provide a way for a program to recover from these errors rather than terminating.

An alternative is to have the `push` and `pop` functions return a `bool` value to indicate whether or not they succeeded. `push` currently has a `void` return type, so it would be easy to modify it to return `true` if the `push` operation succeeds and `false` if the stack is full. Modifying the `pop` function would be more difficult, since `pop` currently returns the value that was popped. However, if `pop` were to return a *pointer* to this value, instead of the value itself, then `pop` could return `NULL` to indicate that the stack is empty.

`assert` macro ➤*24.1*

A final comment about error handling: The C standard library contains a parameterized macro named `assert` that can terminate a program if a specified condition isn't satisfied. We could use calls of this macro as replacements for the `if` statements and calls of `terminate` that currently appear in the stack ADT.

Generic ADTs

Midway through Section 19.4, we improved the stack ADT by making it easier to change the type of items stored in a stack—all we had to do was modify the definition of the `Item` type. It's still somewhat of a nuisance to do so; it would be nicer if a stack could accommodate items of any type, without the need to modify the `stack.h` file. Also note that our stack ADT suffers from a serious flaw: a program can't create two stacks whose items have different types. It's easy to create multiple stacks, but those stacks must have items with identical types. To allow stacks with different item types, we'd have to make copies of the stack ADT's header file and source file and modify one set of files so that the `Stack` type and its associated functions have different names.

What we'd like to have is a single "generic" stack type from which we could create a stack of integers, a stack of strings, or any other stack that we might need. There are various ways to create such a type in C, but none are completely satisfactory. The most common approach uses `void *` as the item type, which allows arbitrary pointers to be pushed and popped. With this technique, the `stack-ADT.h` file would be similar to our original version; however, the prototypes of the `push` and `pop` functions would have the following appearance:

```
void push(Stack s, void *p);
void *pop(Stack s);
```

`pop` returns a pointer to the item popped from the stack; if the stack is empty, it returns a null pointer.

There are two disadvantages to using `void *` as the item type. One is that this approach doesn't work for data that can't be represented in pointer form. Items could be strings (which are represented by a pointer to the first character in the string) or dynamically allocated structures but not basic types such as `int` and `double`. The other disadvantage is that error checking is no longer possible. A stack that stores `void *` items will happily allow a mixture of pointers of different types; there's no way to detect an error caused by pushing a pointer of the wrong type.

ADTs in Newer Languages

The problems that we've just discussed are dealt with much more cleanly in newer C-based languages, such as C++, Java, and C#. Name clashes are prevented by defining function names within a ***class.*** A stack ADT would be represented by a `Stack` class; the stack functions would belong to this class, and would only be recognized by the compiler when applied to a `Stack` object. These languages have a feature known as ***exception handling*** that allows functions such as `push` and `pop` to "throw" an exception when they detect an error condition. Code in the client can then deal with the error by "catching" the exception. C++, Java, and C# also provide special features for defining generic ADTs. In C++, for example, we would define a stack ***template,*** leaving the item type unspecified.

Q & A

Q: You said that C wasn't designed for writing large programs. Isn't UNIX a large program? [p. 483]

A: Not at the time C was designed. In a 1978 paper, Ken Thompson estimated that the UNIX kernel was about 10,000 lines of C code (plus a small amount of assembler). Other components of UNIX were of comparable size; in another 1978 paper, Dennis Ritchie and colleagues put the size of the PDP-11 C compiler at 9660 lines. By today's standards, these are indeed small programs.

Q: Are there any abstract data types in the C library?

A: Technically there aren't, but a few come close, including the `FILE` type (defined in `<stdio.h>`). Before performing an operation on a file, we must declare a variable of type `FILE *`:

FILE type ➤22.1

```
FILE *fp;
```

The `fp` variable will then be passed to various file-handling functions.

Programmers are expected to treat `FILE` as an abstraction. It's not necessary to know what a `FILE` is in order to use the `FILE` type. Presumably `FILE` is a structure type, but the C standard doesn't even guarantee that. In fact, it's better not to know too much about how `FILE` values are stored, since the definition of the `FILE` type can (and often does) vary from one C compiler to another.

Of course, we can always look in the `stdio.h` file and see what a `FILE` is. Having done so, there's nothing to prevent us from writing code to access the internals of a `FILE`. For example, we might discover that `FILE` is a structure with a member named `bsize` (the file's buffer size):

```
typedef struct {
  ...
  int bsize;   /* buffer size */
  ...
} FILE;
```

Once we know about the `bsize` member, there's nothing to prevent us from accessing the buffer size for a particular file:

```
printf("Buffer size: %d\n", fp->bsize);
```

Doing so isn't a good idea, however, because other C compilers might store the buffer size under a different name, or keep track of it in some entirely different way. Changing the `bsize` member is an even worse idea:

```
fp->bsize = 1024;
```

Unless we know all the details about how files are stored, this is a dangerous thing to do. Even if we *do* know the details, they may change with a different compiler or the next release of the same compiler.

Q: What other incomplete types are there besides incomplete structure types? [p. 492]

A: One of the most common incomplete types occurs when an array is declared with no specified size:

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

After this declaration (which we first encountered in Section 15.2), `a` has an incomplete type, because the compiler doesn't know `a`'s length. Presumably `a` is defined in another file within the program; that definition will supply the missing length. Another incomplete type occurs in declarations that specify no length for an array but provide an initializer:

```
int a[] = {1, 2, 3};
```

In this example, the array `a` initially has an incomplete type, but the type is completed by the initializer.

Declaring a union tag without specifying the members of the union also creates an incomplete type. Flexible array members (a C99 feature) have an incomplete type. Finally, `void` is an incomplete type. The `void` type has the unusual property that it can never be completed, thus making it impossible to declare a variable of this type.

C99

flexible array members ➤ 17.9

Q: What other restrictions are there on the use of incomplete types? [p. 492]

A: The `sizeof` operator can't be applied to an incomplete type (not surprisingly, since the size of an incomplete type is unknown). A member of a structure or union (other than a flexible array member) can't have an incomplete type. Similarly, the elements of an array can't have an incomplete type. Finally, a parameter in a function definition can't have an incomplete type (although this is allowed in a function *declaration*). The compiler "adjusts" each array parameter in a function definition so that it has a pointer type, thus preventing it from having an incomplete type.

Exercises

Section 19.1

1. A ***queue*** is similar to a stack, except that items are added at one end but removed from the other in a ***FIFO*** (first-in, first-out) fashion. Operations on a queue might include:

   Inserting an item at the end of the queue
   Removing an item from the beginning of the queue
   Returning the first item in the queue (without changing the queue)
   Returning the last item in the queue (without changing the queue)
   Testing whether the queue is empty

   Write an interface for a queue module in the form of a header file named `queue.h`.

Section 19.2

W 2. Modify the `stack2.c` file to use the `PUBLIC` and `PRIVATE` macros.

3. (a) Write an array-based implementation of the queue module described in Exercise 1. Use three integers to keep track of the queue's status, with one integer storing the position of the first empty slot in the array (used when an item is inserted), the second storing the position of the next item to be removed, and the third storing the number of items in the queue. An insertion or removal that would cause either of the first two integers to be incremented past the end of the array should instead reset the variable to zero, thus causing it to "wrap around" to the beginning of the array.

(b) Write a linked-list implementation of the queue module described in Exercise 1. Use two pointers, one pointing to the first node in the list and the other pointing to the last node. When an item is inserted into the queue, add it to the end of the list. When an item is removed from the queue, delete the first node in the list.

Section 19.3

W 4. (a) Write an implementation of the `Stack` type, assuming that `Stack` is a structure containing a fixed-length array.

(b) Redo the `Stack` type, this time using a linked-list representation instead of an array. (Show both `stack.h` and `stack.c`.)

5. Modify the `queue.h` header of Exercise 1 so that it defines a `Queue` type, where `Queue` is a structure containing a fixed-length array (see Exercise 3(a)). Modify the functions in `queue.h` to take a `Queue *` parameter.

Section 19.4

6. (a) Add a `peek` function to `stackADT.c`. This function will have a parameter of type `Stack`. When called, it returns the top item on the stack but doesn't modify the stack.

(b) Repeat part (a), modifying `stackADT2.c` this time.

(c) Repeat part (a), modifying `stackADT3.c` this time.

7. Modify `stackADT2.c` so that a stack automatically doubles in size when it becomes full. Have the `push` function dynamically allocate a new array that's twice as large as the old one and then copy the stack contents from the old array to the new one. Be sure to have `push` deallocate the old array once the data has been copied.

Programming Projects

1. Modify Programming Project 1 from Chapter 10 so that it uses the stack ADT described in Section 19.4. You may use any of the implementations of the ADT described in that section.

2. Modify Programming Project 6 from Chapter 10 so that it uses the stack ADT described in Section 19.4. You may use any of the implementations of the ADT described in that section.

3. Modify the `stackADT3.c` file of Section 19.4 by adding an `int` member named `len` to the `stack_type` structure. This member will keep track of how many items are currently stored in a stack. Add a new function named `length` that has a `Stack` parameter and returns the value of the `len` member. (Some of the existing functions in `stackADT3.c` will need to be modified as well.) Modify `stackclient.c` so that it calls the `length` function (and displays the value that it returns) after each operation that modifies a stack.

4. Modify the `stackADT.h` and `stackADT3.c` files of Section 19.4 so that a stack stores values of type `void *`, as described in Section 19.5; the `Item` type will no longer be used. Modify `stackclient.c` so that it stores pointers to strings in the `s1` and `s2` stacks.

5. Starting from the `queue.h` header of Exercise 1, create a file named `queueADT.h` that defines the following `Queue` type:

   ```
   typedef struct queue_type *Queue;
   ```

   `queue_type` is an incomplete structure type. Create a file named `queueADT.c` that contains the full definition of `queue_type` as well as definitions for all the functions in `queue.h`. Use a fixed-length array to store the items in a queue (see Exercise 3(a)). Create a file named `queueclient.c` (similar to the `stackclient.c` file of Section 19.4) that creates two queues and performs operations on them. Be sure to provide `create` and `destroy` functions for your ADT.

6. Modify Programming Project 5 so that the items in a queue are stored in a dynamically allocated array whose length is passed to the `create` function.

7. Modify Programming Project 5 so that the items in a queue are stored in a linked list (see Exercise 3(b)).

20 Low-Level Programming

A programming language is low level when its programs require attention to the irrelevant.

Previous chapters have described C's high-level, machine-independent features. Although these features are adequate for many applications, some programs need to perform operations at the bit level. Bit manipulation and other low-level operations are especially useful for writing systems programs (including compilers and operating systems), encryption programs, graphics programs, and programs for which fast execution and/or efficient use of space is critical.

Section 20.1 covers C's bitwise operators, which provide easy access to both individual bits and bit-fields. Section 20.2 then shows how to declare structures that contain bit-fields. Finally, Section 20.3 describes how certain ordinary C features (type definitions, unions, and pointers) can help in writing low-level programs.

Some of the techniques described in this chapter depend on knowledge of how data is stored in memory, which can vary depending on the machine and the compiler. Relying on these techniques will most likely make a program nonportable, so it's best to avoid them unless absolutely necessary. If you do need them, try to limit their use to certain modules in your program; don't spread them around. And, above all, be sure to document what you're doing!

20.1 Bitwise Operators

C provides six ***bitwise operators***, which operate on integer data at the bit level. We'll discuss the two bitwise shift operators first, followed by the four other bitwise operators (bitwise complement, bitwise *and*, bitwise exclusive *or*, and bitwise inclusive *or*).

Bitwise Shift Operators

The bitwise shift operators can transform the binary representation of an integer by shifting its bits to the left or right. C provides two shift operators, which are shown in Table 20.1.

Table 20.1
Bitwise Shift Operators

Symbol	*Meaning*
<<	left shift
>>	right shift

The operands for << and >> may be of any integer type (including `char`). The integer promotions are performed on both operands; the result has the type of the left operand after promotion.

The value of `i << j` is the result when the bits in `i` are shifted left by `j` places. For each bit that is "shifted off" the left end of `i`, a zero bit enters at the right. The value of `i >> j` is the result when `i` is shifted right by `j` places. If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed. If `i` is a negative number, the result is implementation-defined; some implementations add zeros at the left end, while others preserve the sign bit by adding ones.

portability tip *For portability, it's best to perform shifts only on unsigned numbers.*

The following examples illustrate the effect of applying the shift operators to the number 13. (For simplicity, these examples—and others in this section—use short integers, which are typically 16 bits.)

```
unsigned short i, j;

i = 13;         /* i is now 13 (binary 0000000000001101) */
j = i << 2;     /* j is now 52 (binary 0000000000110100) */
j = i >> 2;     /* j is now  3 (binary 0000000000000011) */
```

As these examples show, neither operator modifies its operands. To modify a variable by shifting its bits, we'd use the compound assignment operators <<= and >>=:

```
i = 13;         /* i is now 13 (binary 0000000000001101) */
i <<= 2;        /* i is now 52 (binary 0000000000110100) */
i >>= 2;        /* i is now 13 (binary 0000000000001101) */
```

⚠ The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises. For example, `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

Table 20.2 lists the remaining bitwise operators.

Table 20.2
Other Bitwise Operators

Symbol	*Meaning*
~	bitwise complement
&	bitwise *and*
^	bitwise exclusive *or*
|	bitwise inclusive *or*

The ~ operator is unary; the integer promotions are performed on its operand. The other operators are binary; the usual arithmetic conversions are performed on their operands.

The ~, &, ^, and | operators perform Boolean operations on all bits in their operands. The ~ operator produces the complement of its operand, with zeros replaced by ones and ones replaced by zeros. The & operator performs a Boolean *and* operation on all corresponding bits in its two operands. The ^ and | operators are similar (both perform a Boolean *or* operation on the bits in their operands); however, ^ produces 0 whenever both operands have a 1 bit, whereas | produces 1.

Don't confuse the *bitwise* operators & and | with the *logical* operators && and ||. The bitwise operators sometimes produce the same results as the logical operators, but they're not equivalent.

The following examples illustrate the effect of the ~, &, ^, and | operators:

```
unsigned short i, j, k;

i = 21;      /* i is now    21 (binary 0000000000010101) */
j = 56;      /* j is now    56 (binary 0000000000111000) */
k = ~i;      /* k is now 65514 (binary 1111111111101010) */
k = i & j;   /* k is now    16 (binary 0000000000010000) */
k = i ^ j;   /* k is now    45 (binary 0000000000101101) */
k = i | j;   /* k is now    61 (binary 0000000000111101) */
```

The value shown for `~i` is based on the assumption that an `unsigned short` value occupies 16 bits.

The ~ operator deserves special mention, since we can use it to help make even low-level programs more portable. Suppose that we need an integer whose bits are all 1. The preferred technique is to write `~0`, which doesn't depend on the number of bits in an integer. Similarly, if we need an integer whose bits are all 1 except for the last five, we could write `~0x1f`.

Each of the ~, &, ^, and | operators has a different precedence:

Highest: ~
&
^
Lowest: |

As a result, we can combine these operators in expressions without having to use parentheses. For example, we could write `i & ~j | k` instead of `(i & (~j)) | k` and `i ^ j & ~k` instead of `i ^ (j & (~k))`. Of course, it doesn't hurt to use parentheses to avoid confusion.

The precedence of &, ^, and | is lower than the precedence of the relational and equality operators. Consequently, statements like the following one won't have the desired effect:

table of operators ➤ *Appendix A*

```
if (status & 0x4000 != 0) ...
```

Instead of testing whether `status & 0x4000` isn't zero, this statement will evaluate `0x4000 != 0` (which has the value 1), then test whether the value of `status & 1` isn't zero.

The compound assignment operators &=, ^=, and |= correspond to the bitwise operators &, ^, and |:

```
i = 21;    /* i is now 21 (binary 0000000000010101) */
j = 56;    /* j is now 56 (binary 0000000000111000) */
i &= j;    /* i is now 16 (binary 0000000000010000) */
i ^= j;    /* i is now 40 (binary 0000000000101000) */
i |= j;    /* i is now 56 (binary 0000000000111000) */
```

Using the Bitwise Operators to Access Bits

When we do low-level programming, we'll often need to store information as single bits or collections of bits. In graphics programming, for example, we may want to squeeze two or more pixels into a single byte. Using the bitwise operators, we can extract or modify data that's stored in a small number of bits.

Let's assume that `i` is a 16-bit `unsigned short` variable. Let's see how to perform the most common single-bit operations on `i`:

- ***Setting a bit.*** Suppose that we want to set bit 4 of `i`. (We'll assume that the leftmost—or ***most significant***—bit is numbered 15 and the least significant is numbered 0.) The easiest way to set bit 4 is to *or* the value of `i` with the constant `0x0010` (a "mask" that contains a 1 bit in position 4):

  ```
  i = 0x0000;          /* i is now 0000000000000000 */
  i |= 0x0010;         /* i is now 0000000000010000 */
  ```

  More generally, if the position of the bit is stored in the variable `j`, we can use a shift operator to create the mask:

idiom
```
i |= 1 << j;          /* sets bit j */
```

For example, if `j` has the value 3, then `1 << j` is `0x0008`.

- ***Clearing a bit.*** To clear bit 4 of `i`, we'd use a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff;          /* i is now 0000000011111111 */
i &= ~0x0010;        /* i is now 0000000011101111 */
```

Using the same idea, we can easily write a statement that clears a bit whose position is stored in a variable:

idiom
```
i &= ~(1 << j);      /* clears bit j */
```

- ***Testing a bit.*** The following `if` statement tests whether bit 4 of `i` is set:

```
if (i & 0x0010) …    /* tests bit 4 */
```

To test whether bit `j` is set, we'd use the following statement:

idiom
```
if (i & 1 << j) …    /* tests bit j */
```

To make working with bits easier, we'll often give them names. For example, suppose that we want bits 0, 1, and 2 of a number to correspond to the colors blue, green, and red, respectively. First, we define names that represent the three bit positions:

```
#define BLUE  1
#define GREEN 2
#define RED   4
```

Setting, clearing, and testing the `BLUE` bit would be done as follows:

```
i |= BLUE;           /* sets BLUE bit   */
i &= ~BLUE;          /* clears BLUE bit */
if (i & BLUE) …      /* tests BLUE bit  */
```

It's also easy to set, clear, or test several bits at time:

```
i |= BLUE | GREEN;           /* sets BLUE and GREEN bits   */
i &= ~(BLUE | GREEN);        /* clears BLUE and GREEN bits */
if (i & (BLUE | GREEN)) …    /* tests BLUE and GREEN bits  */
```

The `if` statement tests whether either the `BLUE` bit *or* the `GREEN` bit is set.

Using the Bitwise Operators to Access Bit-Fields

Dealing with a group of several consecutive bits (a ***bit-field***) is slightly more complicated than working with single bits. Here are examples of the two most common bit-field operations:

- ***Modifying a bit-field.*** Modifying a bit-field requires a bitwise *and* (to clear the bit-field), followed by a bitwise *or* (to store new bits in the bit-field). The following statement shows how we might store the binary value 101 in bits 4–6 of the variable `i`:

```
i = i & ~0x0070 | 0x0050;   /* stores 101 in bits 4-6 */
```

The & operator clears bits 4–6 of `i`; the | operator then sets bits 6 and 4. Notice that `i |= 0x0050` by itself wouldn't always work: it would set bits 6 and 4 but not change bit 5. To generalize the example a little, let's assume that the variable `j` contains the value to be stored in bits 4–6 of `i`. We'll need to shift `j` into position before performing the bitwise *or*:

```
i = (i & ~0x0070) | (j << 4);   /* stores j in bits 4-6 */
```

The | operator has lower precedence than & and <<, so we can drop the parentheses if we wish:

```
i = i & ~0x0070 | j << 4;
```

- ***Retrieving a bit-field.*** When the bit-field is at the right end of a number (in the least significant bits), fetching its value is easy. For example, the following statement retrieves bits 0–2 in the variable `i`:

```
j = i & 0x0007;             /* retrieves bits 0-2 */
```

If the bit-field isn't at the right end of of `i`, then we can first shift the bit-field to the end before extracting the field using the & operator. To extract bits 4–6 of `i`, for example, we could use the following statement:

```
j = (i >> 4) & 0x0007;   /* retrieves bits 4-6 */
```

PROGRAM XOR Encryption

One of the simplest ways to encrypt data is to exclusive-*or* (XOR) each character with a secret key. Suppose that the key is the & character. If we XOR this key with the character z, we'll get the \ character (assuming that we're using the ASCII character set):

ASCII character set ➤Appendix E

```
   00100110  (ASCII code for &)
XOR 01111010  (ASCII code for z)
   01011100  (ASCII code for \)
```

To decrypt a message, we just apply the same algorithm. In other words, by encrypting an already-encrypted message, we'll recover the original message. If we XOR the & character with the \ character, for example, we'll get the original character, z:

```
   00100110  (ASCII code for &)
XOR 01011100  (ASCII code for \)
   01111010  (ASCII code for z)
```

The following program, `xor.c`, encrypts a message by XORing each character with the & character. The original message can be entered by the user or read from a file using input redirection; the encrypted message can be viewed on the screen or saved in a file using output redirection. For example, suppose that the file

input and output redirection ➤22.1

msg contains the following lines:

```
Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
                --Johann Kaspar Lavater (1741-1801)
```

To encrypt the msg file, saving the encrypted message in newmsg, we'd use the following command:

```
xor <msg >newmsg
```

newmsg will now contain these lines:

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
              --lINGHH mGUVGT jGPGRCT (1741-1801)
```

To recover the original message, we'd use the command

```
xor <newmsg
```

which will display it on the screen.

As the example shows, our program won't change some characters, including digits. XORing these characters with & would produce invisible control characters, which could cause problems with some operating systems. In Chapter 22, we'll see how to avoid problems when reading and writing files that contain control characters. Until then, we'll play it safe by using the `isprint` function to make sure that both the original character and the new (encrypted) character are printing characters (i.e., not control characters). If either character fails this test, we'll have the program write the original character instead of the new character.

isprint function ➤23.5

Here's the finished program, which is remarkably short:

xor.c

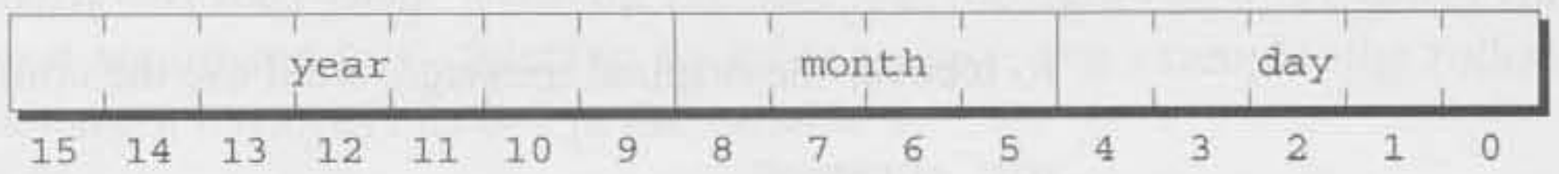
```
/* Performs XOR encryption */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

int main(void)
{
  int orig_char, new_char;

  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (isprint(orig_char) && isprint(new_char))
      putchar(new_char);
    else
      putchar(orig_char);
  }

  return 0;
}
```

20.2 Bit-Fields in Structures

Although the techniques of Section 20.1 allow us to work with bit-fields, these techniques can be tricky to use and potentially confusing. Fortunately, C provides an alternative: declaring structures whose members represent bit-fields.

Q&A

As an example, let's look at how the MS-DOS operating system (often just called DOS) stores the date at which a file was created or last modified. Since days, months, and years are small numbers, storing them as normal integers would waste space. Instead, DOS allocates only 16 bits for a date, with 5 bits for the day, 4 bits for the month, and 7 bits for the year:



Using bit-fields, we can define a C structure with an identical layout:

```
struct file_date {
  unsigned int day: 5;
  unsigned int month: 4;
  unsigned int year: 7;
};
```

The number after each member indicates its length in bits. Since the members all have the same type, we can condense the declaration if we want:

```
struct file_date {
  unsigned int day: 5, month: 4, year: 7;
};
```

The type of a bit-field must be either `int`, `unsigned int`, or `signed int`. Using `int` is ambiguous; some compilers treat the field's high-order bit as a sign bit, but others don't.

portability tip *Declare all bit-fields to be either `unsigned int` or `signed int`.*

C99

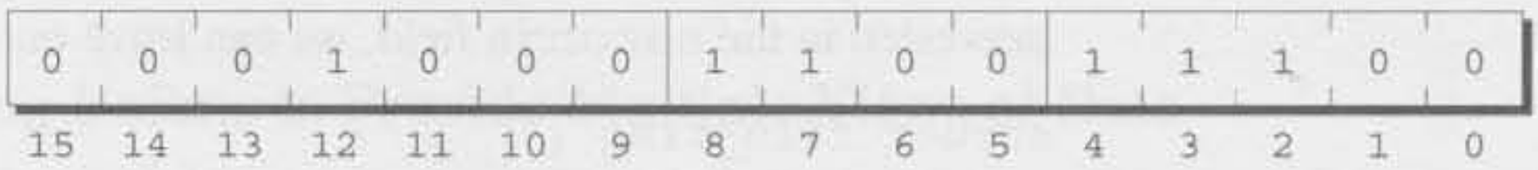
In C99, bit-fields may also have type `_Bool`. C99 compilers may allow additional bit-field types.

We can use a bit-field just like any other member of a structure, as the following example shows:

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 8;      /* represents 1988 */
```

Note that the `year` member is stored relative to 1980 (the year the world began,

according to Microsoft). After these assignments, the `fd` variable will have the following appearance:



We could have used the bitwise operators to accomplish the same effect; using these operators might even make the program a little faster. However, having a readable program is usually more important than gaining a few microseconds.

Bit-fields do have one restriction that doesn't apply to other members of a structure. Since bit-fields don't have addresses in the usual sense, C doesn't allow us to apply the address operator (`&`) to a bit-field. Because of this rule, functions such as `scanf` can't store data directly in a bit-field:

```
scanf("%d", &fd.day);    /*** WRONG ***/
```

Of course, we can always use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

How Bit-Fields Are Stored

Let's take a close look at how a compiler processes the declaration of a structure that has bit-field members. As we'll see, the C standard allows the compiler considerable latitude in choosing how it stores bit-fields.

The rules concerning how the compiler handles bit-fields depend on the notion of "storage units." The size of a storage unit is implementation-defined; typical values are 8 bits, 16 bits, and 32 bits. As it processes a structure declaration, the compiler packs bit-fields one by one into a storage unit, with no gaps between the fields, until there's not enough room for the next field. At that point, some compilers skip to the beginning of the next storage unit, while others split the bit-field across the storage units. (Which one occurs is implementation-defined.) The order in which bit-fields are allocated (left to right or right to left) is also implementation-defined.

Our `file_date` example assumes that storage units are 16 bits long. (An 8-bit storage unit would also be acceptable, provided that the compiler splits the `month` field across two storage units.) We also assume that bit-fields are allocated from right to left (with the first bit-field occupying the low-order bits).

C allows us to omit the name of any bit-field. Unnamed bit-fields are useful as "padding" to ensure that other bit fields are properly positioned. Consider the time associated with a DOS file, which is stored in the following way:

```
struct file_time {
  unsigned int seconds: 5;
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

(You may be wondering how it's possible to store the seconds—a number between 0 and 59—in a field with only 5 bits. Well, DOS cheats: it divides the number of seconds by 2, so the `seconds` member is actually between 0 and 29.) If we're not interested in the `seconds` field, we can leave out its name:

```
struct file_time {
  unsigned int : 5;           /* not used */
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

The remaining bit-fields will be aligned as if the `seconds` field were still present.

Another trick that we can use to control the storage of bit-fields is to specify 0 as the length of an unnamed bit-field:

```
struct s {
  unsigned int a: 4;
  unsigned int : 0;     /* 0-length bit-field */
  unsigned int b: 8;
};
```

A 0-length bit-field is a signal to the compiler to align the following bit-field at the beginning of a storage unit. If storage units are 8 bits long, the compiler will allocate 4 bits for the `a` member, skip 4 bits to the next storage unit, and then allocate 8 bits for `b`. If storage units are 16 bits long, the compiler will allocate 4 bits for `a`, skip 12 bits, and then allocate 8 bits for `b`.

20.3 Other Low-Level Techniques

Some of the language features that we've covered in previous chapters are used often in low-level programming. To wrap up this chapter, we'll take a look at several important examples: defining types that represent units of storage, using unions to bypass normal type-checking, and using pointers as addresses. We'll also cover the `volatile` type qualifier, which we avoided discussing in Section 18.3 because of its low-level nature.

Defining Machine-Dependent Types

Since the `char` type—by definition—occupies one byte, we'll sometimes treat characters as bytes, using them to store data that's not necessarily in character form. When we do so, it's a good idea to define a `BYTE` type:

```
typedef unsigned char BYTE;
```

Depending on the machine, we may want to define additional types. The x86 architecture makes extensive use of 16-bit words, so the following definition would be useful for that platform:

```
typedef unsigned short WORD;
```

We'll use the `BYTE` and `WORD` types in later examples.

Using Unions to Provide Multiple Views of Data

Although unions can be used in a portable way—see Section 16.4 for examples—they're often used in C for an entirely different purpose: viewing a block of memory in two or more different ways.

Here's a simple example based on the `file_date` structure described in Section 20.2. Since a `file_date` structure fits into two bytes, we can think of any two-byte value as a `file_date` structure. In particular, we could view an `unsigned short` value as a `file_date` structure (assuming that short integers are 16 bits long). The following union allows us to easily convert a short integer to a file date or vice versa:

```
union int_date {
  unsigned short i;
  struct file_date fd;
};
```

With the help of this union, we could fetch a file date from disk as two bytes, then extract its `month`, `day`, and `year` fields. Conversely, we could construct a date as a `file_date` structure, then write it to disk as a pair of bytes.

As an example of how we might use the `int_date` union, here's a function that, when passed an `unsigned short` argument, prints it as a file date:

```
void print_date(unsigned short n)
{
  union int_date u;

  u.i = n;
  printf("%d/%d/%d\n", u.fd.month, u.fd.day, u.fd.year + 1980);
}
```

Using unions to allow multiple views of data is especially useful when working with registers, which are often divided into smaller units. x86 processors, for example, have 16-bit registers named AX, BX, CX, and DX. Each of these registers can be treated as two 8-bit registers. AX, for example, is divided into registers named AH and AL. (The H and L stand for "high" and "low.")

When writing low-level applications for x86-based computers, we may need variables that represent the contents of the AX, BX, CX, and DX registers. We want access to both the 16- and 8-bit registers; at the same time, we need to take their relationships into account (a change to AX affects both AH and AL; changing AH or AL modifies AX). The solution is to set up two structures, one containing members that correspond to the 16-bit registers, and the other containing members that match the 8-bit registers. We then create a union that encloses the two structures:

```
union {
  struct {
    WORD ax, bx, cx, dx;
  } word;
  struct {
    BYTE al, ah, bl, bh, cl, ch, dl, dh;
  } byte;
} regs;
```

The members of the `word` structure will be overlaid with the members of the `byte` structure; for example, `ax` will occupy the same memory as `al` and `ah`. And that, of course, is exactly what we wanted. Here's an example showing how the `regs` union might be used:

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %hx\n", regs.word.ax);
```

Changing `ah` and `al` affects `ax`, so the output will be

```
AX: 1234
```

Note that the `byte` structure lists `al` before `ah`, even though the AL register is the "low" half of AX and AH is the "high" half. Here's the reason. When a data item consists of more than one byte, there are two logical ways to store it in memory: with the bytes in the "natural" order (with the leftmost byte stored first) or with the bytes in reverse order (the leftmost byte is stored last). The first alternative is called ***big-endian;*** the second is known as ***little-endian.*** C doesn't require a specific byte ordering, since that depends on the CPU on which a program will be executed. Some CPUs use the big-endian approach and some use the little-endian approach. What does this have to do with the `byte` structure? It turns out that x86 processors assume that data is stored in little-endian order, so the first byte of `regs.word.ax` is the low byte.

Q&A

We don't normally need to worry about byte ordering. However, programs that deal with memory at a low level must be aware of the order in which bytes are stored (as the `regs` example illustrates). It's also relevant when working with files that contain non-character data.

Be careful when using unions to provide multiple views of data. Data that is valid in its original format may be invalid when viewed as a different type, causing unexpected problems.

Using Pointers as Addresses

We saw in Section 11.1 that a pointer is really some kind of memory address, although we usually don't need to know the details. When we do low-level programming, however, the details matter.

An address often has the same number of bits as an integer (or long integer). Creating a pointer that represents a specific address is easy: we just cast an integer into a pointer. For example, here's how we might store the address 1000 (hex) in a pointer variable:

```
BYTE *p;

p = (BYTE *) 0x1000;    /* p contains address 0x1000 */
```

PROGRAM

Viewing Memory Locations

Our next program allows the user to view segments of computer memory; it relies on C's willingness to allow an integer to be used as a pointer. Most CPUs execute programs in "protected mode," however, which means that a program can access only those portions of memory that belong to the program. This prevents a program from accessing (or changing) memory that belongs to another application or to the operating system itself. As a result, we'll only be able to use our program to view areas of memory that have been allocated for use by the program itself. Going outside these regions will cause the program to crash.

The `viewmemory.c` program begins by displaying the address of its own `main` function as well as the address of one of its variables. This will give the user a clue as to which areas of memory can be probed. The program next prompts the user to enter an address (in the form of a hexadecimal integer) plus the number of bytes to view. The program then displays a block of bytes of the chosen length, starting at the specified address.

Bytes are displayed in groups of 10 (except for the last group, which may have fewer than 10 bytes). The address of a group of bytes is displayed at the beginning of a line, followed by the bytes in the group (displayed as hexadecimal numbers); followed by the same bytes displayed as characters (just in case the bytes happen to represent characters, as some of them may). Only printing characters (as determined by the `isprint` function) will be displayed; other characters will be shown as periods.

We'll assume that `int` values are stored using 32 bits and that addresses are also 32 bits long. Addresses are displayed in hexadecimal, as is customary.

viewmemory.c

```
/* Allows the user to view regions of computer memory */

#include <ctype.h>
#include <stdio.h>

typedef unsigned char BYTE;

int main(void)
{
  unsigned int addr;
  int i, n;
  BYTE *ptr;

  printf("Address of main function: %x\n", (unsigned int) main);
  printf("Address of addr variable: %x\n", (unsigned int) &addr);
```

```
    printf("\nEnter a (hex) address: ");
    scanf("%x", &addr);
    printf("Enter number of bytes to view: ");
    scanf("%d", &n);

    printf("\n");
    printf(" Address              Bytes                 Characters\n");
    printf(" -------  ----------------------------  ----------\n");

    ptr = (BYTE *) addr;
    for (; n > 0; n -= 10) {
      printf("%8X  ", (unsigned int) ptr);
      for (i = 0; i < 10 && i < n; i++)
        printf("%.2X ", *(ptr + i));
      for (; i < 10; i++)
        printf("   ");
      printf(" ");
      for (i = 0;  i < 10 && i < n; i++) {
        BYTE ch = *(ptr + i);
        if (!isprint(ch))
          ch = '.';
        printf("%c", ch);
      }
      printf("\n");
      ptr += 10;
    }

    return 0;
}
```

The program is complicated somewhat by the possibility that the value of `n` isn't a multiple of 10, so there may be fewer than 10 bytes in the last group. Two of the `for` statements are controlled by the condition `i < 10 && i < n`. This condition causes the loops to execute 10 times or `n` times, whichever is smaller. There's also a `for` statement that compensates for any missing bytes in the last group by displaying three spaces for each missing byte. That way, the characters that follow the last group of bytes will align properly with the character groups on previous lines.

The `%X` conversion specifier used in this program is similar to `%x`, which was discussed in Section 7.1. The difference is that `%X` displays the hexadecimal digits A, B, C, D, E, and F as upper-case letters; `%x` displays them in lower case.

Here's what happened when I compiled the program using GCC and tested it on an x86 system running Linux:

```
Address of main function: 804847c
Address of addr variable: bff41154

Enter a (hex) address: 8048000
Enter number of bytes to view: 40

 Address              Bytes                 Characters
 -------  ----------------------------  ----------
 8048000  7F 45 4C 46 01 01 01 00 00 00  .ELF......
 804800A  00 00 00 00 00 00 02 00 03 00  ..........
 8048014  01 00 00 00 C0 83 04 08 34 00  ........4.
 804801E  00 00 C0 0A 00 00 00 00 00 00  ..........
```

I asked the program to display 40 bytes starting at address 8048000, which precedes the address of the `main` function. Note the 7F byte followed by bytes representing the letters E, L, and F. These four bytes identify the format (ELF) in which the executable file was stored. ELF (Executable and Linking Format) is widely used by UNIX systems, including Linux. 8048000 is the default address at which ELF executables are loaded on x86 platforms.

Let's run the program again, this time displaying a block of bytes that starts at the address of the `addr` variable:

```
Address of main function: 804847c
Address of addr variable: bfec5484

Enter a (hex) address: bfec5484
Enter number of bytes to view: 64

 Address           Bytes                       Characters
 -------  -----------------------------------  ----------
BFEC5484  84 54 EC BF B0 54 EC BF F4 6F  .T...T...o
BFEC548E  68 00 34 55 EC BF C0 54 EC BF  h.4U...T..
BFEC5498  08 55 EC BF E3 3D 57 00 00 00  .U...=W...
BFEC54A2  00 00 A0 BC 55 00 08 55 EC BF  ....U..U..
BFEC54AC  E3 3D 57 00 01 00 00 00 34 55  .=W.....4U
BFEC54B6  EC BF 3C 55 EC BF 56 11 55 00  ..<U..V.U.
BFEC54C0  F4 6F 68 00                    .oh.
```

None of the data stored in this region of memory is in character form, so it's a bit hard to follow. However, we do know one thing: the `addr` variable occupies the first four bytes of this region. When reversed, these bytes form the number BFEC5484, the address entered by the user. Why the reversal? Because x86 processors store data in little-endian order, as we saw earlier in this section.

The `volatile` Type Qualifier

On some computers, certain memory locations are "volatile"; the value stored at such a location can change as a program is running, even though the program itself isn't storing new values there. For example, some memory locations might hold data coming directly from input devices.

The `volatile` type qualifier allows us to inform the compiler if any of the data used in a program is volatile. `volatile` typically appears in the declaration of a pointer variable that will point to a volatile memory location:

```
volatile BYTE *p;   /* p will point to a volatile byte */
```

To see why `volatile` is needed, suppose that `p` points to a memory location that contains the most recent character typed at the user's keyboard. This location is volatile: its value changes each time the user enters a character. We might use the following loop to obtain characters from the keyboard and store them in a buffer array:

```
while (buffer not full) {
  wait for input;
  buffer[i] = *p;
  if (buffer[i++] == '\n')
    break;
}
```

A sophisticated compiler might notice that this loop changes neither `p` nor `*p`, so it could optimize the program by altering it so that `*p` is fetched just once:

```
store *p in a register;
while (buffer not full) {
  wait for input;
  buffer[i] = value stored in register;
  if (buffer[i++] == '\n')
    break;
}
```

The optimized program will fill the buffer with many copies of the same character—not exactly what we had in mind. Declaring that `p` points to volatile data avoids this problem by telling the compiler that `*p` must be fetched from memory each time it's needed.

Q & A

Q: What do you mean by saying that the `&` and `|` operators sometimes produce the same results as the `&&` and `||` operators, but not always? [p. 511]

A: Let's compare `i & j` with `i && j` (similar remarks apply to `|` and `||`). As long as `i` and `j` have the value 0 or 1 (in any combination), the two expressions will have the same value. However, if `i` and `j` should have other values, the expressions may not always match. If `i` is 1 and `j` is 2, for example, then `i & j` has the value 0 (`i` and `j` have no corresponding 1 bits), while `i && j` has the value 1. If `i` is 3 and `j` is 2, then `i & j` has the value 2, while `i && j` has the value 1.

Side effects are another difference. Evaluating `i & j++` *always* increments `j` as a side effect, whereas evaluating `i && j++` *sometimes* increments `j`.

Q: Who cares how DOS stores file dates? Isn't DOS dead? [p. 516]

A: For the most part, yes. However, there are still plenty of files created years ago whose dates are stored in the DOS format. In any event, DOS file dates are a good example of how bit-fields are used.

Q: Where do the terms "big-endian" and "little-endian" come from? [p. 520]

A: In Jonathan Swift's novel *Gulliver's Travels,* the fictional islands of Lilliput and Blefuscu are perpetually at odds over whether to open boiled eggs on the big end or the little end. The choice is arbitrary, of course, just like the order of bytes in a data item.

Exercises

Section 20.1

*1. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `unsigned short` variables.

(a)
```
i = 8; j = 9;
printf("%d", i >> 1 + j >> 1);
```

(b)
```
i = 1;
printf("%d", i & ~i);
```

(c)
```
i = 2; j = 1; k = 0;
printf("%d", ~i & j ^ k);
```

(d)
```
i = 7; j = 8; k = 9;
printf("%d", i ^ j & k);
```

W 2. Describe a simple way to "toggle" a bit (change it from 0 to 1 or from 1 to 0). Illustrate the technique by writing a statement that toggles bit 4 of the variable `i`.

*3. Explain what effect the following macro has on its arguments. You may assume that the arguments have the same type.

```
#define M(x,y)  ((x)^=(y),(y)^=(x),(x)^=(y))
```

W 4. In computer graphics, colors are often stored as three numbers, representing red, green, and blue intensities. Suppose that each number requires eight bits, and we'd like to store all three values in a single long integer. Write a macro named `MK_COLOR` with three parameters (the red, green, and blue intensities). `MK_COLOR` should return a `long` in which the last three bytes contain the red, green, and blue intensities, with the red value as the last byte and the green value as the next-to-last byte.

5. Write macros named `GET_RED`, `GET_GREEN`, and `GET_BLUE` that, when given a color as an argument (see Exercise 4), return its 8-bit red, green, and blue intensities.

W 6. (a) Use the bitwise operators to write the following function:

```
unsigned short swap_bytes(unsigned short i);
```

`swap_bytes` should return the number that results from swapping the two bytes in `i`. (Short integers occupy two bytes on most computers.) For example, if `i` has the value `0x1234` (00010010 00110100 in binary), then `swap_bytes` should return `0x3412` (00110100 00010010 in binary). Test your function by writing a program that reads a number in hexadecimal, then writes the number with its bytes swapped:

```
Enter a hexadecimal number (up to four digits): 1234
Number with bytes swapped: 3412
```

Hint: Use the `%hx` conversion to read and write the hex numbers.

(b) Condense the `swap_bytes` function so that its body is a single statement.

7. Write the following functions:

```
unsigned int rotate_left(unsigned int i, int n);
unsigned int rotate_right(unsigned int i, int n);
```

`rotate_left` should return the result of shifting the bits in `i` to the left by `n` places, with the bits that were "shifted off" moved to the right end of `i`. (For example, the call

`rotate_left(0x12345678, 4)` should return `0x23456781` if integers are 32 bits long.) `rotate_right` is similar, but it should "rotate" bits to the right instead of the left.

W 8. Let `f` be the following function:

```
unsigned int f(unsigned int i, int m, int n)
{
  return (i >> (m + 1 - n)) & ~(~0 << n);
}
```

(a) What is the value of `~(~0 << n)`?

(b) What does this function do?

9. (a) Write the following function:

```
int count_ones(unsigned char ch);
```

`count_ones` should return the number of 1 bits in `ch`.

(b) Write the function in part (a) without using a loop.

10. Write the following function:

```
unsigned int reverse_bits(unsigned int n);
```

`reverse_bits` should return an unsigned integer whose bits are the same as those in `n` but in reverse order.

11. Each of the following macros defines the position of a single bit within an integer:

```
#define SHIFT_BIT 1
#define CTRL_BIT  2
#define ALT_BIT   4
```

The following statement is supposed to test whether any of the three bits have been set, but it never displays the specified message. Explain why the statement doesn't work and show how to fix it. Assume that `key_code` is an `int` variable.

```
if (key_code & (SHIFT_BIT | CTRL_BIT | ALT_BIT) == 0)
  printf("No modifier keys pressed\n");
```

12. The following function supposedly combines two bytes to form an unsigned short integer. Explain why the function doesn't work and show how to fix it.

```
unsigned short create_short(unsigned char high_byte,
                            unsigned char low_byte)
{
  return high_byte << 8 + low_byte;
}
```

*13. If `n` is an `unsigned int` variable, what effect does the following statement have on the bits in `n`?

```
n &= n - 1;
```

Hint: Consider the effect on `n` if this statement is executed more than once.

Section 20.2

W 14. When stored according to the IEEE floating-point standard, a `float` value consists of a 1-bit sign (the leftmost—or most significant—bit), an 8-bit exponent, and a 23-bit fraction, in that order. Design a structure type that occupies 32 bits, with bit-field members corresponding to the sign, exponent, and fraction. Declare the bit-fields to have type `unsigned int`. Check the manual for your compiler to determine the order of the bit-fields.

*15. (a) Assume that the variable `s` has been declared as follows:

```
struct {
  int flag: 1;
} s;
```

With some compilers, executing the following statements causes 1 to be displayed, but with other compilers, the output is –1. Explain the reason for this behavior.

```
s.flag = 1;
printf("%d\n", s.flag);
```

(b) How can this problem be avoided?

Section 20.3

16. Starting with the 386 processor, x86 CPUs have 32-bit registers named EAX, EBX, ECX, and EDX. The second half (the least significant bits) of these registers is the same as AX, BX, CX, and DX, respectively. Modify the `regs` union so that it includes these registers as well as the older ones. Your union should be set up so that modifying EAX changes AX and modifying AX changes the second half of EAX. (The other new registers will work in a similar fashion.) You'll need to add some "dummy" members to the `word` and `byte` structures, corresponding to the other half of EAX, EBX, ECX, and EDX. Declare the type of the new registers to be `DWORD` (double word), which should be defined as `unsigned long`. Don't forget that the x86 architecture is little-endian.

Programming Projects

1. Design a union that makes it possible to view a 32-bit value as either a `float` or the structure described in Exercise 14. Write a program that stores 1 in the structure's sign field, 128 in the exponent field, and 0 in the fraction field, then prints the `float` value stored in the union. (The answer should be –2.0 if you've set up the bit-fields correctly.)