

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**.

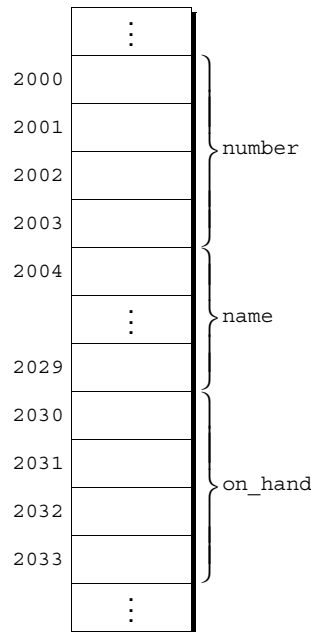
Declaring Structure Variables

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:

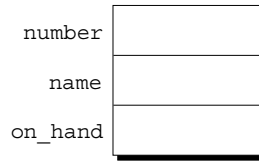
```
struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1, part2;
```

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.

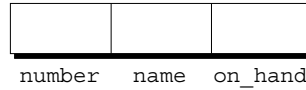
The members of a structure are stored in memory in the order in which they're declared. In order to show what the `part1` variable looks like in memory, let's assume that (1) `part1` is located at address 2000, (2) integers occupy four bytes, (3) `NAME_LEN` has the value 25, and (4) there are no gaps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

<code>number</code>	528
<code>name</code>	Disk drive
<code>on_hand</code>	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

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.

FILE type ►22.1

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.

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.

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.

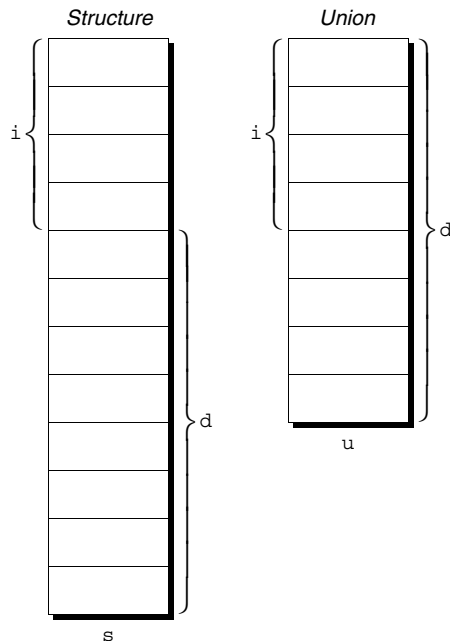
To illustrate the basic properties of unions, let's declare a union variable, `u`, with two members:

```
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.

protecting header files ► 15.2

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.

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). **C99** 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:

```
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. **C99** Incidentally, C99 also allows trailing commas in compound literals.

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:

```
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:

- Compute the area of `r`.
- 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.
- 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.)
- 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;

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:
- Compute the area of `s`.
 - 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.)
 - 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

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?
- An enumeration constant may represent any integer specified by the programmer.
 - Enumeration constants have exactly the same properties as constants created using `#define`.
 - Enumeration constants have the values 0, 1, 2, ... by default.
 - All constants in an enumeration must have different values.
 - Enumeration constants may be used as integers in expressions.
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)?
- `b = FALSE;`
  - `b = i;`
  - `b++;`
  - `i = b;`
  - `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.
- Using the types from part (a), define a structure type named `Square` that can store both the type of a piece and its color.
  - Using the `Square` type from part (b), declare an  $8 \times 8$  array named `board` that can store the entire contents of a chessboard.
  - 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.