4 Pointers and Structures

In the last chapter, we used the memory map to study how arrays and strings work. Writing out a memory map is often useful during program design. During debugging, it can be invaluable. In this chapter, we extend these ideas to pointers and structures. All variables have an address in memory, much the same way that all houses and businesses have a street address in the real world. However, the organization of buildings and streets becomes clearer by looking at a map. It tends to be easier to explore an unknown city or to reach a destination using a map. In much the same way, it tends to be easier to design code for a problem involving pointers or to debug the code using a memory map. Pointers (and to some degree, structures) can be the most difficult tool to master in the C programming language. The goal of this chapter is to increase the proficiency of the reader with these tools through a deeper understanding of how they work.

4.1 • Pointers

A pointer is a construct used to store an address of a variable. We declare a variable to be a pointer-type variable by preceding its name with the asterisk symbol (*). For example:

```
char c,*cp;
int i,*ip;
float f,*fp;
double d,*dp;
```

The variables `c`, `i`, `f`, and `d` are normal variables, each holding a different type of value. The variables `cp`, `ip`, `fp`, and `dp` are all pointers, each holding an address.

How big is each of these variables? From previous chapters, we know that a `char` is 1 byte, an `int` and a `float` are 4 bytes each (although they use the bits differently), and a `double` is 8 bytes. How many bytes does a pointer use?

To answer that question, consider what a pointer holds: an address. A pointer needs enough bits to store all the possible addresses on the computing system. How many addresses are there? At the time of this writing, a common computing system can have a maximum of 4 GB of memory. It takes 32 bits to store roughly 4 billion different addresses, one for each byte of memory. In fact, this is precisely why such a computing system is limited to 4 GB of memory; it is because it has a 32-bit address bus connecting the CPU to memory. Throughout this book,[1] we have been assuming a 32-bit architecture to explain memory concepts.

Knowing how many bytes a pointer variable requires, we can now draw a memory map for our example above:

Label	Address	Value
`c`	400	
`cp`	401–404	
`i`	405–408	
`ip`	409–412	
`f`	413–416	
`fp`	417–420	
`d`	421–428	
`dp`	429–432	

An address is an address, regardless of what resides at that address. All addresses require 4 bytes. Therefore all pointers, regardless of the type of variable "pointed to," require 4 bytes.

The use of pointer variables involves two symbols: `&` and `*`. The ampersand symbol (`&`) indicates the "address of" a variable, and the asterisk symbol (`*`) indicates "at the address given by" a variable. For example:

```
cp=&c;
ip=&i;
*ip=42;
```

The memory map for this code can be filled in as follows:

1. Please see the discussion in Section 2.1.1.

Label	Address	Value
c	400	
cp	401–404	400
i	405–408	42
ip	409–412	405
f	413–416	
fp	417–420	
d	421–428	
dp	429–432	

The first two lines store the addresses of the variables `c` and `i` into the pointer variables `cp` and `ip`. The last line looks at the address stored in `ip`, which is 405, and places the value 42 at that address. Address 405 is where the variable `i` resides, so in effect `i` is the variable with a new value.

A pointer can hold the address of any variable, including a cell in an array. For example:

```
char ca[3],*cp;
ca[1]=3;
cp=&(ca[1]);
*cp=7;
```

The memory map for this code may be written as:

Label	Address	Value
ca[0]	400	
ca[1]	401	3 7
ca[2]	402	
cp	403–406	401

When writing code using multiple symbols, it is a good practice to use parentheses to emphasize the order of operations. Although it is possible to write `&ca[1]`, and it means the same thing as `&(ca[1])`, the latter is preferable because of its clarity. This becomes increasingly true as variables are used in greater complexity. An advanced programmer may not need to be reminded of the order of operations, but including the parentheses never hurts.

An address can be printed out using the printf() function in a few different ways. Since it is a 4-byte whole number, it can be printed as an int (using %i

or %d), but this interprets the highest bit as a sign bit, so that the value may be negative. It is easier to read if it is printed as an unsigned int (using %u) so that the number is always positive. The printf() function also provides the %p identifier for pointers, to display the address in hexadecimal. For example:

```
char a,*b;
a=7; b=&a;
printf("%d %u %p  value=%d\n",b,b,b,*b);
```

The output of this code is as follows:

```
-1073764873 3221202423 0xbfffa5f7  value=7
```

The unsigned base 10 display of an address is often the easiest to read, but the hexadecimal display of an address is sometimes preferred because it is the easiest to translate to binary (each hexadecimal digit converts to four binary digits). Notice that the address, in unsigned base 10, is a much larger number than the range we have been using (the 400's). For purposes of code design, it is convenient to use small numbers for addresses when writing a memory map, but during debugging, it may be necessary to work with large actual address values like these.

4.1.1 Pointer Arithmetic

There are two different uses for the * symbol,[2] one during variable *declaration*, and one during variable *usage*. This can lead to some confusion. The distinction is that in the first case, the * symbol is indicating the variable type (pointer), while in the second case, the * symbol is using the pointer to place a value at another address. It is natural to ask why we use the complicated notation

```
char *cp;
```

to declare a variable, instead of something simple like

```
pointer cp;
```

After all, if we have keywords to define the four basic data types, why not a keyword to define this "fifth data type" that stores an address? The answer is pointer arithmetic. When adding or subtracting amounts from an address, pointer arithmetic acts in quantities of bytes equal to the size of the thing referenced. For example, consider the following code:

2. Of course, the * symbol is also used for multiplication; here we are strictly talking about pointers.

```
char ca[3],*cp;
int ia[3],*ip;
cp=&(ca[0]);
ip=&(ia[0]);
```

The memory map for this code may be written as:

Label	Address	Value
ca[0]	400	
ca[1]	401	
ca[2]	402	
cp	403–406	400
ia[0]	407–410	
ia[1]	411–414	
ia[2]	415–418	
ip	419–422	407

Now consider the following lines of code involving pointer arithmetic:

```
*(cp+2)=8;     /* cp + 2 what? */
*(ip+2)=33;    /* ip + 2 what? */
```

In each line of code, the * symbol indicates that a value is to be placed at the given address. But at what address? In the first line, we have the value of "cp+2" as the address. The value of cp is 400. Adding 2 gives an address of 402. Looking at the memory map, the variable ca[2] resides at that address, so this seems to make sense. Now consider the second line, where we have the value of "ip+2" as the address. The value of ip is 407. Adding 2 gives an address of 409, which is the midpoint of the variable ia[0]. This does not make sense.

Pointer arithmetic uses the units of the type of variable "pointed to." Those units are set during the variable declaration. Since cp is a pointer to char, its units are 1-byte increments. Since ip is a pointer to int, its units are 4-byte increments. Using the correct units, we can determine that:

```
cp+2 = cp+2 (1 byte units) = 400+2 = 402
ip+2 = ip+2 (4 byte units) = 407+8 = 415
```

The resulting memory map may be written as follows:

Label	Address	Value
ca[0]	400	
ca[1]	401	
ca[2]	402	8
cp	403–406	400
ia[0]	407–410	
ia[1]	411–414	
ia[2]	415–418	33
ip	419–422	407

Notice that using pointer arithmetic, the offsets match the indices of the arrays. This is the whole point. Pointers and arrays can be used interchangeably; in fact, they are often the same thing. This concept is discussed more in the next section.

4.2 • Using Pointers

Pointers are a difficult tool to master. They are the source of many coding errors and bugs, leading to a number of flaws and security problems. Why then do we use them? This section discusses some of the most common reasons for using pointers.

4.2.1 Passing Values Back from a Function

The most common reason for using pointers is to pass values back from a function. Consider the following program:

```
#include <stdio.h>

int division(int numerator, int denominator,
             int *dividend, int *remainder)
{
printf("address stored in dividend: %u\n",dividend);
printf("address stored in remainder: %u\n",remainder);
if (denominator == 0)
  return(0);
*dividend=numerator/denominator;
*remainder=numerator%denominator;
```

```
return(1);
}

main()
{
int     x,y,d,r;
x=9;
y=2;
printf("address of d: %u\n",&d);
printf("address of r: %u\n",&r);
division(x,y,&d,&r);
printf("%d/%d = %d with %d remainder\n",x,y,d,r);
printf("x=%d\n",x);
}
```

First, we look at the label and address (the size of each variable) columns of the memory map. This may be written as follows:

Label	Address	Value
numerator	400–403	
denominator	404–407	
dividend	408–411	
remainder	412–415	
x	700–703	
y	704–707	
d	708–711	
r	712–715	

The parameters of the division() function are variables, so they must be included in the memory map. They should be treated just like variables declared inside a function. The addresses used for the main() function (700's) have been somewhat separated from those used for the division() function (400's), simply to help organization. In reality, they might be right next to each other in memory, or far apart, or anywhere in between. The size of every variable is 4 bytes, because they are all either int or pointer variables.

When the code executes, the values 9 and 2 go into x and y, and then the function call happens. What does that do? It makes a copy of all the parameters, and places them in the local memory locations for the function. At this point, the memory map may be written as:

Label	Address	Value
numerator	400–403	9
denominator	404–407	2
dividend	408–411	708
remainder	412–415	712
x	700–703	9
y	704–707	2
d	708–711	
r	712–715	

Notice that the values of x and y are copied to numerator and denominator, but that it is the addresses of d and r that are copied to dividend and remainder. After this, the function code is executed. Using the pointer to dereference the address, the results are placed in the memory locations for d and r. At this point, the memory map may be written as:

Label	Address	Value
numerator	400–403	9
denominator	404–407	2
dividend	408–411	708
remainder	412–415	712
x	700–703	9
y	704–707	2
d	708–711	4
r	712–715	1

Notice that those results are never in any local memory location for the division() function, they are stored only in the original main() function variables. This is why pointers are necessary. It allows a function to be called to do some work, putting the results in variables residing in the original function. Variables like x and y are known as "call by value" parameters, while those like d and r are known as "call by reference" parameters. These phrases refer to how a variable value is passed to a function.

Suppose we added the following line of code to the end of the division() function:

```
numerator=7;
```

Looking at the memory map, we can see why this will have no effect upon the variable x in the main() function. Changing the value at address 400, where numerator resides, has no effect upon the value at address 700, where the variable x resides. This shows why we *must* use a pointer to pass a result back from a function call.

4.2.2 Pointers and Arrays

The second most common use of a pointer is when an array is used. Arrays are pointers, insofar as they are both addresses. We saw in the last chapter that an array name is a label for the starting address of the array. We saw in the last section that pointer arithmetic works similarly to array indexing. We will now put these ideas together and show how array and pointer syntax are interchangeable. Consider the following program:

```
#include <stdio.h>

main()
{
double  array[5];
double  *d_ptr;
double  value;
int     i,offset;

for (i=0; i<5; i++)
  array[i]=(double)i+10.0;    /* fill array with #'s */

d_ptr=&(array[0]);            /* set up pointer */

while (1)
  {
  printf("Address(hex)\tAddress(base10)\tValue\n");
  for (i=0; i<5; i++)
    printf("%p\t%u\t%lf\n",&(array[i]),&(array[i]),array[i]);
  printf("Enter offset value (0 0 to quit): ");
  scanf("%d %lf",&offset,&value);
  if (offset == 0  &&  value == 0.0)
    break;                    /* break out of loop */
  if (offset < 0  ||  offset > 4)
    {
    printf("Offset out of bounds\n");
```

```
        continue;                /* go back to start of loop */
        }
          /* three ways to do the same thing: */
      array[offset]=value;       /* using array syntax */
      *(d_ptr+offset)=value;     /* using pointer syntax */
      *(array+offset)=value;     /* using mixed syntax */
      }
}
```

This program sets up an array of five doubles, and then enters a loop. The user can select an array index and new value, which the program then places into the given index. The program also checks for valid array indices (within the bounds 0 to 4), and uses the index/value pair 0/0 to quit.

The last three lines of code are the most interesting. They show how we can access the same memory using array indexing, pointer arithmetic, or even a mix of both. All three of those lines of code do the exact same thing. We can see this by looking at the memory map:

Address label(s)	Label	Address	Value
array &(array[0])	array[0]	400–407	10.0
	array[1]	408–415	11.0
	array[2]	416–423	12.0
	array[3]	424–431	13.0
	array[4]	432–439	14.0
	d_ptr	440–443	400
	value	444–451	
	i	452–455	~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5
	offset	456–459	

The value at address 400 can be accessed through its original name `array[0]`, or through the pointer value of `d_ptr`, or through mixed notation using a pointer to the original address label `array`. It can even be accessed using another mixed notation based on the other original address label `&(array[0])`:

```
*(&(array[0])+offset)=value;
```

There is one important difference between using an address label, such as `array`, and a pointer variable, such as `d_ptr`. The former has a fixed value that cannot be changed. It refers to the address 400, and that cannot be changed. However, the

value stored in the pointer variable can be changed at any time. This is sometimes used as an argument to support the idea that pointers and arrays are different. However, looking at the memory underneath, we can see that both can be used to access an ordered block of memory. This is why they can be interchanged and are often considered the same thing.

4.2.3 Dynamic Memory Allocation

Variables are normally given space through static memory allocation. This means that the size (in bytes) of each of the variables is known before the program runs. Upon starting execution of the program, the operating system (O/S) finds a place in memory for all the global variables in the program (in an area called the data segment). As each function is entered, including main(), space for its variables and parameter values are found (in an area called the stack). Without going into the details of the implementation of these memory areas, it is important to note that the entire time that the program is running, the size of a statically allocated variable does not change.

Sometimes a program does not know how much memory it needs prior to execution. For example, in reading a line of text from a file, the program might not know how big a string is needed to store all the words before the particular file has been selected, and read. In this case, a programmer might decide to declare a string of sufficient size to store any possible line of text. This is wasteful and perhaps, in some cases, impossible. The maximum size of something to be read might not be known beforehand. The alternative is to use dynamic memory allocation.

Dynamic memory allocation uses a pointer variable to request memory from the O/S while the program is running. The basic function call for dynamic memory allocation is `malloc()`, and works as follows:

```
#include <stdlib.h>  /* header file for malloc() */
double  *a;          /* pointer variable */
a = (double *) malloc  (40);
/*   ^^^^^^^^  ^^^^^^   ^^                 */
/*   typecast  request  how many bytes?    */
/*             to O/S                      */
```

The typecast identifies the type of pointer arithmetic expected for the address returned from malloc() (see Section 4.2.2 on pointer arithmetic). The malloc() function call is the request to the O/S for memory. It takes one parameter, which is the number of bytes needed. In this example, we request 40 bytes, which is 5 doubles (8 bytes per double).

How does it work? One of the primary jobs of the O/S is to manage memory. A malloc() call asks the O/S for a chunk of memory of the given size. The chunk of memory is obtained from an area of memory called the heap. Compared to the other memory areas (e.g., the stack or the data segment), the heap can normally support larger variables. In general, a program can dynamically acquire much larger chunks of memory than can be obtained using static variable declarations. The address of that chunk of memory, if available, is returned. The memory map for this code may be written as follows:

Label	Address	Value
a	400–403	10000
[DM]	10000–10039	

Dynamically allocated memory does not have an existing label, so we use [DM] to temporarily provide it a name. In order to give it a name that our program can use, we must use either pointer or array syntax based upon the variable a. Either syntax can be used to address, or label, different parts of the 40 bytes requested. For example, we can write the following code:

```
a[0]=8;
*(a+2)=3;
a[3]=9;
```

The memory map for this code may be written as:

Label(s)	Address	Value
a	400–403	10000
*(a+0) a[0]	10000–10007	8
*(a+1) a[1]	10008–10015	
*(a+2) a[2]	10016–10023	3
*(a+3) a[3]	10024–10031	9
*(a+4) a[4]	10032–10039	

Every cell of memory in the chunk of bytes has both a pointer-based label and an array-based label and can be accessed using either syntax. It is quite common to see a pointer variable used for dynamic memory allocation and then accessed using array syntax.

Every time a function ends, its statically declared memory is released from the stack and returned to the O/S. This includes the ending of the main function,

which is the end of the program. Dynamic memory allocation is different. Because the allocation can be made at any point in a program, the program must also be responsible for releasing the memory and returning it to the O/S. This is done by calling the `free()` function:

```
free(a);
```

Failure to call the free() function appropriately may result in a program losing track of the memory it uses. Sometimes this is referred to as a memory leak. For example, consider the following code:

```
double *a;
a=(double *)malloc(70);
a=(double *)malloc(300);  /* memory leak */
```

The program requests 70 bytes from the O/S, and then 300 bytes, each time storing the address of the bytes in a. But the second request overwrites the address of the first request. What happens to those 70 bytes? They are still reserved for use by the program, but the program has lost any record of the address where they reside! This is a common programming mistake and can lead to a program crashing.

There are several variants on the malloc() function that request bytes in slightly different ways. A common variant is `calloc()`, which in addition to requesting the bytes also initializes the value of every byte to zero. For example:

```
double *a;
a=(double *)calloc(70,sizeof(double));
```

The calloc() function takes two arguments; the first is the number of cells of memory, and the second is the size of each cell. The `sizeof()` operator is a convenient tool for denoting the number of bytes in a data type. It is evaluated by the compiler at compile time to find the size of a variable or variable type. It is often used with structures and complex data type combinations, which we will see more of in the next section. For this example, the sizeof() operator returns 8, which is the size of a double, so the total number of bytes requested is 560.

4.2.4 Double Pointers

In the last section, we saw how a chunk of dynamically allocated memory can be accessed as an array. In effect, this is like creating a variable-length, one-dimensional array, where the length is determined while the program is running. This concept can be extended to multidimensional arrays by using multiple

pointers. For example, we can use a double pointer to act like a two-dimensional array. Consider the following code, which declares a double pointer:

```
double  **ptr;
```

It may be understood by rewriting it, emphasizing the order of applying each of the * operators:

```
double  *(*ptr);
```

We know that `*ptr` is an address of a double. The * symbol means "at the address given by;" therefore, `*(*ptr)` must mean "the double at the address given by the address given by." How big, in bytes, is a double pointer? It's still just an address, so it is 4 bytes. All addresses are 4 bytes, even an address of an address.

The following code demonstrates using a double pointer like a two-dimensional array:

```
double **m;
m=(double **)calloc(2,sizeof(double *));
m[0]=(double *)calloc(3,sizeof(double));
m[1]=(double *)calloc(2,sizeof(double));
m[0][1]=6.3;
m[1][0]=-2.8;
```

The first calloc() function call asks for enough bytes to store two pointers (addresses), or 8 bytes total. Each of these pointers is then used to store the address of a number of doubles (3 doubles in the second calloc(), and 2 doubles in the third calloc()). Once all the memory is allocated, it may be accessed as though it were a two-dimensional array. The memory map for this code may be written as:

Label(s)	Address	Value
m	400–403	10000
*(m+0) m[0]	10000–10003	10008
*(m+1) m[1]	10004–10007	10032
((m+0)+0) m[0][0]	10008–10015	
((m+0)+1) m[0][1]	10016–10023	6.3
((m+0)+2) m[0][2]	10024–10031	
((m+1)+0) m[1][0]	10032–10039	-2.8
((m+1)+1) m[1][1]	10040–10047	

The variable m holds a double pointer, or the address of a list of addresses (10000). The list of addresses is m[0] and m[1], each of which holds an address of a list of doubles (10008 and 10032).

Another common use of double pointers is the passing of a pointer variable to a function. This is necessary when a function needs to dynamically allocate memory and then pass that memory back to the calling function. The following code demonstrates this use of a double pointer:

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

int integers(int listsize,int **list)
{
int     i;
*list=(int *)malloc(listsize*sizeof(int));
if (*list == NULL)
  return(0);
for (i=0; i<listsize; i++)
  (*list)[i]=10+i;  /* mixed array/pointer syntax */
return(1);
}

int main(int argc, char *argv[])
{
int     *numbers;
int     i;
i=integers(3,&numbers);
for (i=0; i<3; i++)
  printf("%d\n",numbers[i]);
}
```

The integers() function dynamically allocates space for a list of ints and then puts values in each cell. One of its parameters indicates how large a list to create. At the completion of execution of the program, the memory map may be written as:

Label(s)	Address	Value
listsize	400–403	3
list	404–407	708
i	408–411	~~0~~ ~~1~~ ~~2~~ 3
argc	700–703	

(continued)

Label(s)		Address	Value
argv		704–707	
numbers		708–711	10000
i		712–715	~~0~~ ~~1~~ ~~2~~ 3
(*list)[0]	numbers[0]	10000–10003	10
(*list)[1]	numbers[1]	10004–10007	11
(*list)[2]	numbers[2]	10008–10011	12

This memory map brings up a few interesting points. First, it includes argc and argv, which are variables local to the main() function. The latter (argv) is a double pointer and in memory looks like the examples just shown. Second, it includes two variables named i. Each of these is local to a different function, which is why the same name can be used twice. Third, the syntax (*list)[0] shows a mixed pointer/array reference to the double pointer. In most cases, it is easier to read code that sticks to a pure pointer or pure array syntax. For example, the line

```
*(list)[i]=10+i;
```

may be rewritten as

```
*((*list)+i)=10+i;
```

4.3 • Structures

A structure is a construct used to group a set of variables together under one name. The first step in using a structure is to declare its organization. For example:

```
struct person {    /* "person" is name for structure type */
char   first[32];  /* first field is array of char */
char   last[32];   /* second field is array of char */
int    year;       /* third field is int */
double ppg;        /* fourth field is double */
};                 /* ending ; to end definition */
```

This code does not create a variable. There are no bytes of storage named yet. Instead, this code creates a *template* for a new variable type called struct person. One can think of a struct person as being a similar construct to an int or double. It is a name for a data type, not a name for a variable. In this example,

the `struct person` consists of two arrays of 32 `char`, an `int`, and a `double`, for a total of 76 bytes.

We can use this definition of a `struct person` to declare a variable as follows:

```
struct person    teacher;
```

We now have a variable named `teacher` that provides 76 bytes of storage. Each part of the variable is called a field and is accessed using the period (`.`) symbol. For example:

```
teacher.year=2006;
teacher.ppg=10.4;
strcpy(teacher.first,"Adam");
strcpy(teacher.last,"Hoover");
```

The memory map for this code may be written as:

Address label(s)	Label(s)	Address	Value
`teacher.first`	`teacher  teacher.first[0]`	400	`'A'`
	`teacher.first[1]`	401	`'d'`
	`teacher.first[2]`	402	`'a'`
	`teacher.first[3]`	403	`'m'`
	`teacher.first[4]`	404	`'\0'`
	`teacher.first[5]-[31]`	405–431	
`teacher.last`	`teacher.last[0]`	432	`'H'`
	`teacher.last[1]`	433	`'o'`
	`teacher.last[2]`	434	`'o'`
	`teacher.last[3]`	435	`'v'`
	`teacher.last[4]`	436	`'e'`
	`teacher.last[5]`	437	`'r'`
	`teacher.last[6]`	438	`'\0'`
	`teacher.last[7]-[31]`	439–463	
	`teacher.year`	464–467	2005
	`teacher.ppg`	468–475	10.4

There are two labels for the data at address 400, `teacher.first[0]` and `teacher`. The former refers to the `char` (1 byte) at that address, while the latter refers to the `struct person` (76 bytes) at that address. Of course, the latter includes the former as its first byte. It should come as no surprise that a label can refer to multiple bytes starting at a given address. For example, `teacher.ppg` refers to

the double (8 bytes) starting at address 468, not just byte 468. The number of bytes is known by the data type of the label. In the case of `teacher`, the data type is a `struct person`, and so the label refers to 76 bytes.

Having a label that refers to the entire structure is convenient for two purposes. First, it allows assignment statements between structure variables. For example, continuing the code from above:

```
struct person   mailman,teacher;
mailman=teacher;     /* copy entire contents of struct */
```

Assigning a structure to another copies every byte. Of course, the structures must be of the same type. The second convenience is in passing a structure as a parameter to a function. For example:

```
DisplayStats(struct person   Input)
{
printf("%s, %s:  %lf PPG in %d\n",Input.last,Input.first,
                          Input.ppg,Input.year);
}

main()
{
struct person   teacher;
/* .... */
DisplayStats(teacher);
}
```

When the function call to DisplayStats() is made, all 76 bytes in the variable `teacher` are copied to `Input`. These examples demonstrate the whole point of using a structure. It is convenient to group a number of variables (hence, a large number of bytes) under a single variable name.

Looking back at the memory map for `teacher`, from above, we had the following line:

Address label(s)	Label(s)	Address	Value
`teacher.first`	`teacher  teacher.first[0]`	400	`'A'`

Unlike an array variable name, a structure variable name is a label for the values in the given bytes, not for the address. In this manner, a structure variable name acts like a regular (char, int, float, double) variable name. In the example, we see that the label `teacher.first` is a name for address 400, which is the beginning address of an array of char. The label `teacher` is a name for the values of the 76

bytes starting at address 400; it is not another label for address 400. However, we can use the & symbol prior to either regular label to turn it into an address label (for example, `&teacher` or `&(teacher.first[0])`).

4.4 • Using Structures

Structure variables can in many cases be treated exactly like regular variables. For example, they can be used in assignment statements and passed as function parameters, and a field of a structure variable can be used in regular calculations. However, structures can be combined with arrays, pointers, and other structures, making the syntax and usage complicated. A good approach to unraveling a strange combination, in order to design, debug, or understand code, is to examine it at the memory level. The following sections demonstrate this approach.

4.4.1 Arrays and Structures

It is possible to create an array of structs, just like it is possible to have an array of any data type. The template for the structure definition remains unchanged; it still must be declared first. Using our definition of a `struct person` from above, we may write the following code:

```
struct person   class[54];     /* array of "struct person" */

class[0].year=2006;   /* notice where array subscript goes */
class[0].ppg=5.2;
strcpy(class[0].first,"Jane");
strcpy(class[0].last,"Doe");
class[1].first[0]='B'; /* array field in a struct array */
class[1].first[1]='o';
class[1].first[2]='b';
class[1].first[3]=0;
```

The memory map for this code may be written as:

Address label(s)	Label(s)	Address	Value
`class  class[0].first`	`class[0]  class[0].first[0]`	400	`'J'`
	`class[0].first[1]`	401	`'a'`
	`class[0].first[2]`	402	`'n'`
	`class[0].first[3]`	403	`'e'`
	`class[0].first[4]`	404	`'\0'`

(continued)

Address label(s)	Label(s)	Address	Value
	`class[0].first[5]-[31]`	405–431	
`class[0].last`	`class[0].last[0]`	432	`'D'`
	`class[0].last[1]`	433	`'o'`
	`class[0].last[2]`	434	`'e'`
	`class[0].last[3]`	435	`'\0'`
	`class[0].last[4]-[31]`	436–463	
	`class[0].year`	464–467	2006
	`class[0].ppg`	468–475	5.2
`class[1].first`	`class[1] class[1].first[0]`	476	`'B'`
	`class[1].first[1]`	477	`'o'`
	`class[1].first[2]`	478	`'b'`
	`class[1].first[3]`	479	`'\0'`
	`class[1].first[4]-[31]`	480–507	
`class[1].last`	`class[1].last[0]-[31]`	508–539	
	`class[1].year`	540-543	
	`class[1].ppg`	544–551	
	`class[2]`	552–627	
	`class[3]`	628–703	
	`class[4]--[53]`	704–4503	

Back in Chapter 3, we learned that an array name by itself is an address label. This is true regardless of the type of array. Thus, `class`, which is the name for an array of `struct person`, is an address label. So is `class[0].first`, which is the name for an array of char. Both of these refer to address 400. On the other hand, `class[0]` is not the name of an array; it is the name of a cell within the array. Therefore it is not an address label; it is a regular label. So is `class[0].first[0]`. The labels `class[0]` and `class[0].first[0]` do differ in how many bytes are labeled. The former labels 76 bytes as a group, while the latter labels only 1 byte.

4.4.2 Definitions and Scope

Structure variables can be global or local to a function, just like any other variable. Structure definitions can also be global or local. In addition, when making a definition, a variable using that definition can be declared at the same time. Consider the following code:

```
struct fraction {    /* structure definition */
                int     x,y;
                } global;  /* variable using this definition */
                  /* variable outside a function is global */
main()
{
      /* "throw away" structure template (used only once) */
struct {    /* notice there is no name for structure def. */
       char    title[20];
       float   cost;
       } paperback;  /* variable using this definition */
int     n;
struct fraction local;

n=3;
paperback.cost=4.50;
strcpy(paperback.title,"C");
}

some_function()
{
struct fraction x;  /* var. names are independent of fields */

global.x=1;
x.x=2;              /* so "x" can be a var. and field name */
}
```

At the end of a structure definition, one or more variables can be declared that use that definition. Thus, `global` and `paperback` are both variables. The former is a `struct fraction` type variable, while the latter . . . has no name for its type! It is possible to define a structure template but never give that template a name and use it only to define a variable at the end of its definition. This makes it impossible to declare a variable using that particular structure definition at a later time. This sort of thing is rarely done and in general is not recommended. However, a programmer should be able to understand these sorts of tricks, which are best examined using a memory map:

Address label(s)	Label(s)	Address	Value
	`global  global.x`	100–103	`1`
	`global.y`	104–107	
`paperback.title`	`paperback.title[0]`	400	`'C'`
	`paperback.title[1]`	401	`'\0'`

(continued)

Address label(s)	Label(s)	Address	Value
	paperback.title[2]-[19]	402–419	
	paperback.cost	420–423	4.5
	n	424–427	3
	local local.x	428–431	
	local.y	432–435	
	x x.x	700–703	2
	x.y	704–707	

Writing out the memory map enforces an understanding of which things are variables: `fraction` is a structure definition, not a variable name. It does not belong in the memory map. This memory map also shows how a label can be used for both a structure variable (`x`) and a field within that structure (`x.x`). Although they label bytes starting at the same address (700), `x` refers to a `struct fraction` (8 bytes), while `x.x` refers to an `int` (4 bytes).

4.4.3 Nested Structures

Structures can be nested, so that a field within a structure is itself another structure. For example:

```
struct name {
            char   first[32];
            char   last[32];
            };

struct person {
              int           age;
              float         ppg;
              struct name   title;      /* nested structure */
              };            /* "name" must be defined above */

struct person   boss;
boss.age=80;
boss.ppg=0.1;
strcpy(boss.title.first,"Dean");
strcpy(boss.title.last,"Smith");
```

The memory map for this code may be written as:

Address label(s)	Label(s)	Address	Value
	boss boss.age	400–403	80
	boss.ppg	404–407	0.1
boss.title.first	boss.title boss.title.first[0]	408	'D'
	boss.title.first[1]	409	'e'
	boss.title.first[2]	410	'a'
	boss.title.first[3]	411	'n'
	boss.title.first[4]	412	'\0'
	boss.title.first[5]-[31]	413–439	
boss.title.last	boss.title.last[0]	440	'S'
	boss.title.last[1]	441	'm'
	boss.title.last[2]	442	'i'
	boss.title.last[3]	443	't'
	boss.title.last[4]	444	'h'
	boss.title.last[5]	445	'\0'
	boss.title.last[6]-[31]	446–471	

The label `boss` is a name for a `struct person` (72 bytes), while the label `boss.title` is a name for a `struct name` (64 bytes). The other nested structure names, `boss.title.first` and `boss.title.last`, are address labels for arrays of char, referring to addresses 408 and 440, respectively.

4.4.4 Pointers and Structures

The address of a structure variable can be stored in a pointer variable, just like the address of any other type of variable. For example:

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

struct fraction   f[3],*g;
f[0].x=3;
f[0].y=7;
g=&(f[0]);
g++;
```

The memory map for this code may be written as:

Address label(s)	Label(s)	Address	Value
f	f[0] f[0].x	400–403	3
	f[0].y	404–407	7
	f[1] f[1].x	408–411	
	f[1].y	412–415	
	f[2] f[2].x	416–419	
	f[2].y	420–423	
	g	424–427	~~400~~ 408

The variable g holds an address of a `struct fraction`. Using pointer arithmetic, this means that the units of addition for g are 8 bytes. Therefore, `g++` adds 8 to the current value of g.

Continuing from this code example, we can access the bytes of the structure through the pointer variable. There are two different syntaxes available to do this. The following demonstrates each syntax:

```
(*g).x=5;
g->y=11;
```

The first line uses the same syntax as all other types of pointers. The value of g is 408; the * symbol says to go to that address; then the `.x` says to go to that field at the given address. The C language provides a second syntax that accomplishes the same steps. The `->` syntax (hyphen followed by greater-than symbol) means "in the field at the given address." The memory map for this code may be updated as follows:

Address label(s)	Label(s)	Address	Value
f	f[0] f[0].x	400–403	3
	f[0].y	404–407	7
	f[1] f[1].x	408–411	5
	f[1].y	412–415	11
	f[2] f[2].x	416–419	
	f[2].y	420–423	
	g	424–427	~~400~~ 408

A pointer variable for a structure can also be used to dynamically allocate memory. For example:

```
g=(struct fraction *)malloc(sizeof(struct fraction));
```

A `struct fraction` is 8 bytes in size; the `sizeof()` operator is a convenient method to determine the number of bytes needed.[3] By default, the `malloc()` function returns a void pointer (`void *`), which means that the returned value is an address but that the type or size of variable at that address could be anything. This lets the malloc() function be used to allocate space for any type of pointer. It is common practice to typecast the return value from the malloc() function call to make the code easier to read by showing how the pointer is dereferenced. As noted above, the variable g holds a pointer to `struct fraction`, which increments in units of 8 bytes. The memory map after this allocation may be updated as:

Address label(s)	Label(s)	Address	Value
f	f[0] f[0].x	400–403	3
	f[0].y	404–407	7
	f[1] f[1].x	408–411	5
	f[1].y	412–415	11
	f[2] f[2].x	416–419	
	f[2].y	420–423	
	g	424–427	~~400~~ ~~408~~ 10000
	g.x	10000–10003	
	g.y	10004–10007	