CHAPTER 11

Structuring Data

So far, you've learned how to declare and define variables that can hold various types of data, including integers, floating-point values, and characters. You also have the means to create arrays of any of these types and arrays of pointers to memory locations containing data of the types available to you. Although these have proved very useful, there are many applications in which you need even more flexibility.

For instance, suppose you want to write a program that processes data about breeding horses. You need information about each horse such as its name, its date of birth, its coloring, its height, its parentage, and so on. Some items are strings and some are numeric. Clearly, you could set up arrays for each data type and store them quite easily. However, this has limitations—for example, it doesn't allow you to refer to Dobbin's date of birth or Trigger's height particularly easily. You would need to synchronize your arrays by relating data items through a common index. Amazingly, C provides you with a better way of doing this, and that's what I'll discuss in this chapter.

In this chapter you'll learn the following:

- What structures are
- How to declare and define data structures
- How to use structures and pointers to structures
- How you can use pointers as structure members
- How to share memory between variables
- How to define your own data types
- How to write a program that produces bar charts from your data

Data Structures: Using struct

The keyword `struct` enables you to define a collection of variables of various types called a **structure** that you can treat as a single unit. This will be clearer if you see a simple example of a structure declaration:

```
struct horse
{
  int age;
  int height;
} Silver;
```

This example declares a **structure** type called `horse`. This isn't a variable name; it's a new type. This type name is usually referred to as a **structure tag**, or a **tag name**. The naming of the structure tag follows the same rules as for a variable name, which you should be familiar with by now.

■ **Note** It's legal to use the same name for a structure tag name and another variable. However, I don't recommend that you do this because it will make your code confusing and difficult to understand.

The variable names within the structure, `age` and `height`, are called **structure members**. In this case, they're both of type `int`. The members of the structure appear between the braces that follow the `struct` tag name `horse`.

In the example, an instance of the structure, called `Silver`, is declared at the same time that the structure is defined. `Silver` is a variable of type `horse`. Now, whenever you use the variable name `Silver`, it includes both members of the structure: the member `age` and the member `height`.

Let's look at the declaration of a slightly more complicated version of the structure type `horse`:

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
} Dobbin = {
             24, 17, "Dobbin", "Trigger", "Flossie"
           };
```

Any type of variable can appear as a member of a structure, including arrays. As you can see, there are five members to this version of the structure type called `horse`: the integer members `age` and `height`, and the arrays `name`, `father`, and `mother`. Each member declaration is essentially the same as a normal variable declaration, with the type followed by the name and terminated by a semicolon. Note that initialization values can't be placed here because you aren't declaring variables; you're defining members of a type called `horse`. A structure type is a kind of specification or blueprint that can then be used to define variables of that type—in this example, type `horse`.

You define an instance of the structure `horse` after the closing brace of the structure definition. This is the variable `Dobbin`. Initial values are also assigned to the member variables of `Dobbin` in a manner similar to that used to initialize arrays, so initial values *can* be assigned when you define instances of the type `horse`.

In the declaration of the variable `Dobbin`, the values that appear between the final pair of braces apply, in sequence, to the member variable `age` (24), `height` (17), `name` (`"Dobbin"`), `father` (`"Trigger"`), and `mother` (`"Flossie"`). The statement is finally terminated with a semicolon. The variable `Dobbin` now refers to the complete collection of members included in the structure. The memory occupied by the structure `Dobbin` is shown in Figure 11-1, assuming variables of type `int` occupy 4 bytes. You can always find out the amount of memory that's occupied by a structure using the `sizeof` operator.

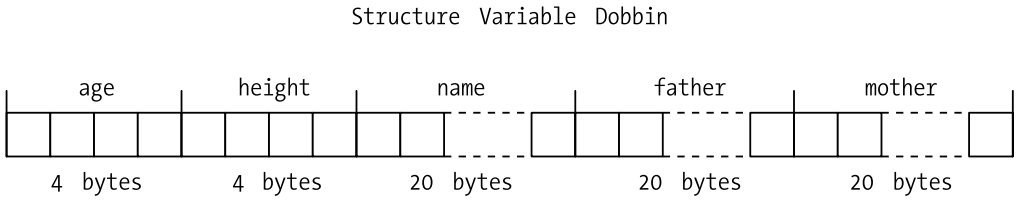

Figure 11-1. *Memory occupied by Dobbin*

Defining Structure Types and Structure Variables

You could have separated the declaration of the structure from the declaration of the structure variable. Instead of the statements you saw previously, you could have written the following:

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};

struct horse Dobbin = {
                        24, 17,"Dobbin", "Trigger", "Flossie"
                      };
```

You now have two separate statements. The first is the definition of the structure tag `horse`, and the second is a declaration of one variable of that type, `Dobbin`. Both the structure definition and the structure variable declaration statements end with a semicolon. The initial values for the members of the `Dobbin` structure tell you that the father of Dobbin is Trigger and his mother is Flossie.

You could also add a third statement to the previous two examples that would define another variable of type `horse`:

```
struct horse Trigger = {
                         30, 15, "Trigger", "Smith", "Wesson"
                       };
```

Now you have a variable `Trigger` that holds the details of the father of `Dobbin`, where it's clear that the ancestors of `Trigger` are `"Smith"` and `"Wesson"`.

Of course, you can also declare multiple structure variables in a single statement. This is almost as easy as declaring variables of one of the standard C types, for example

```
struct horse Piebald, Bandy;
```

declares two variables of type `horse`. The only additional item in the declaration, compared with standard types, is the keyword `struct`. You haven't initialized the values—to keep the statement simple—but in general it's wise to do so.

Accessing Structure Members

Now that you know how to define a structure and declare structure variables, you need to be able to refer to the members of a structure. A structure variable name is *not* a pointer. You need a special syntax to access the members.

You refer to a member of a structure by writing the structure variable name followed by a period, followed by the member variable name. For example, if you found that `Dobbin` had lied about his age and was actually much younger than the initializing value would suggest, you could amend the value by writing this:

```
Dobbin.age = 12;
```

The period between the structure variable name and the member name is actually an operator that is called the **member selection operator**. This statement sets the `age` member of the structure

Dobbin to 12. Structure members are just the same as variables of the same type. You can set their values and use them in expressions in the same way as ordinary variables.

TRY IT OUT: USING STRUCTURES

You can try out what you've learned so far about structures in a simple example that's designed to appeal to horse enthusiasts:

```
/* Program 11.1 Exercising the horse */
#include <stdio.h>

int main(void)
{
  /* Structure declaration */
  struct horse
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
  };

  struct horse My_first_horse;         /* Structure variable declaration   */

  /* Initialize the structure variable from input data */
  printf("Enter the name of the horse: " );
  scanf("%s", My_first_horse.name );            /* Read the horse's name   */

  printf("How old is %s? ", My_first_horse.name );
  scanf("%d", &My_first_horse.age );            /* Read the horse's age    */

  printf("How high is %s ( in hands )? ", My_first_horse.name );
  scanf("%d", &My_first_horse.height );         /* Read the horse's height */

  printf("Who is %s's father? ", My_first_horse.name );
  scanf("%s", My_first_horse.father );          /* Get the father's name   */

  printf("Who is %s's mother? ", My_first_horse.name );
  scanf("%s", My_first_horse.mother );          /* Get the mother's name   */

  /* Now tell them what we know */
  printf("\n%s is %d years old, %d hands high,",
       My_first_horse.name, My_first_horse.age, My_first_horse.height);
  printf(" and has %s and %s as parents.\n", My_first_horse.father,
                                          My_first_horse.mother );
  return 0;
}
```

Depending on what data you key in, you should get output approximating to the following:

```
Enter the name of the horse: Neddy
How old is Neddy? 12
How high is Neddy ( in hands )? 14
Who is Neddy's father? Bertie
Who is Neddy's mother? Nellie

Neddy is 12 years old, 14 hands high, and has Bertie and Nellie as parents.
```

How It Works

The way you reference members of a structure makes it very easy to follow what is going on in this example. You define the structure `horse` with this statement:

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

The structure has two integer members, `age` and `height`, and three `char` array members, `name`, `father`, and `mother`. Because there's just a semicolon following the closing brace, no variables of type `horse` are declared here.

After defining the structure `horse`, you have the following statement:

```
struct horse My_first_horse;          /* Structure variable declaration   */
```

This declares one variable of type `horse`, which is `My_first_horse`. This variable has no initial values assigned in the declaration.

You then read in the data for the `name` member of the structure `My_first_horse` with this statement:

```
scanf("%s", My_first_horse.name );            /* Read the horse's name   */
```

No address of operator (&) is necessary here, because the `name` member of the structure is an array, so you implicitly transfer the address of the first array element to the function `scanf()`. You reference the member by writing the structure name, `My_first_horse`, followed by a period, followed by the name of the member, which is `name`. Other than the notation used to access it, using a structure member is the same as using any other variable.

The next value you read in is for the age of the horse:

```
scanf("%d", &My_first_horse.age );            /* Read the horse's age    */
```

This member is a variable of type `int`, so here you must use the & to pass the address of the structure member.

Note When you use the address of operator (&) for a member of a `struct` object, you place the & in front of the whole reference to the member, not in front of the member name.

The following statements read the data for each of the other members of the structure in exactly the same manner, prompting for the input in each case. Once input is complete, the values read are output to the display as a single line using the following statements:

```
printf("\n\n%s is %d years old, %d hands high,",
       My_first_horse.name, My_first_horse.age, My_first_horse.height);
  printf(" and has %s and %s as parents.", My_first_horse.father,
                                           My_first_horse.mother );
```

The long names that are necessary to refer to the members of the structure tend to make this statement appear complicated, but it's quite straightforward. You have the names of the member variables as the arguments to the function following the first argument, which is the standard sort of format control string that you've seen many times before.

Unnamed Structures

You don't have to give a structure a tag name. When you declare a structure and any instances of that structure in a single statement, you can omit the tag name. In the last example, instead of the structure declaration for type `horse`, followed by the instance declaration for `My_first_horse`, you could have written this statement:

```
struct
{                                          /* Structure declaration and... */
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
} My_first_horse;             /* ...structure variable declaration combined */
```

A serious disadvantage with this approach is that you can no longer define further instances of the structure in another statement. All the variables of this structure type that you want in your program must be defined in the one statement.

Arrays of Structures

The basic approach to keeping horse data is fine as far as it goes. But it will probably begin to be a bit cumbersome by the time you've accumulated 50 or 100 horses. You need a more stable method for handling a lot of horses. It's exactly the same problem that you had with variables, which you solved using an array. And you can do the same here: you can declare a `horse` array.

TRY IT OUT: USING ARRAYS OF STRUCTURES

Let's saddle up and extend the previous example to handle several horses:

```
/* Program 11.2   Exercising the horses */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  struct horse                        /* Structure declaration */
  {
    int age;
    int height;
```

```
 char name[20];
    char father[20];
    char mother[20];
  };

  struct horse My_horses[50];         /* Structure array declaration   */
  int hcount = 0;                     /* Count of the number of horses */
  char test = '\0';                   /* Test value for ending         */

  for(hcount = 0; hcount<50 ; hcount++ )
  {
    printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                          hcount?"nother " : "" );
    scanf(" %c", &test );
    if(tolower(test) == 'n')
       break;

    printf("\nEnter the name of the horse: " );
    scanf("%s", My_horses[hcount].name );  /* Read the horse's name */

    printf("\nHow old is %s? ", My_horses[hcount].name );
    scanf("%d", &My_horses[hcount].age );  /* Read the horse's age  */

    printf("\nHow high is %s ( in hands )? ", My_horses[hcount].name );
    /* Read the horse's height*/
    scanf("%d", &My_horses[hcount].height );

    printf("\nWho is %s's father? ", My_horses[hcount].name );
    /* Get the father's name */
     scanf("%s", My_horses[hcount].father );

    printf("\nWho is %s's mother? ", My_horses[hcount].name );

    /* Get the mother's name  */
    scanf("%s", My_horses[hcount].mother );
  }

  /* Now tell them what we know. */
  for(int i = 0 ; i<hcount ; i++ )
  {
    printf("\n\n%s is %d years old, %d hands high,",
             My_horses[i].name, My_horses[i].age, My_horses[i].height);
    printf(" and has %s and %s as parents.", My_horses[i].father,
                                             My_horses[i].mother );
  }
  return 0;
}
```

The output from this program is a little different from the previous example you saw that dealt with a single horse. The main addition is the prompt for input data for each horse. Once all the data for a few horses has been entered, or if you have the stamina, the data on 50 horses has been entered, the program outputs a summary of all the data that has been read in, one line per horse. The whole mechanism is stable and works very well in the mane (almost an unbridled success, you might say).

How It Works

In this version of equine data processing, you first declare the `horse` structure, and this is followed by the declaration

```
struct horse My_horses[50]; /* Structure array declaration   */
```

This declares the variable `My_horses`, which is an array of 50 `horse` structures. Apart from the keyword `struct`, it's just like any other array declaration.

You then have a `for` loop controlled by the variable `hcount`:

```
for(hcount = 0; hcount<50 ; hcount++ )
{
  ...
}
```

This creates the potential for the program to read in data for up to 50 horses. The loop control variable `hcount` is used to accumulate the total number of `horse` structures entered. The first action in the loop is in these statements:

```
printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                          hcount?"nother " : "" );
scanf(" %c", &test );
if(tolower(test) == 'n')
   break;
```

On each iteration the user is prompted to indicate if he or she wants to enter data for another horse by entering **Y** or **N**. The `printf()` statement for this uses the conditional operator to insert `"nother"` into the output on every iteration after the first. After reading the character that the user enters, using `scanf()`, the `if` statement executes a `break`, which immediately exits from the loop if the response is negative.

The succeeding sequence of `printf()` and `scanf()` statements are much the same as before, but there are two points to note in these. Look at this statement:

```
scanf("%s", My_horses[hcount].name );  /* Read the horse's name */
```

You can see that the method for referencing the member of one element of an array of structures is easier to write than to say! The structure array name has an index in square brackets, to which a period and the member name are appended. If you want to reference the third element of the `name` array for the fourth structure element, you would write `My_horses[3].name[2]`.

Note Of course, the index values start from 0, as with arrays of other types, so the fourth element of the structure array has the index value 3, and the third element of the member array is accessed by the index value 2.

Now look at this statement from the example:

```
scanf("%d", &My_horses[hcount].age );  /* Read the horse's age  */
```

Notice that the arguments to `scanf()` don't need the & for the string array variables, such as `My_horses[hcount].name`, but they *do* require them for the integer arguments `My_horses[hcount].age` and `My_horses[hcount].height`. It's very easy to forget the address of operator when reading values for variables like these.

Don't be misled at this point and think that these techniques are limited to equine applications. They can perfectly well be applied to porcine problems and also to asinine exercises.

Structures in Expressions

A structure member that is one of the built-in types can be used like any other variable in an expression. Using the structure from Program 11.2, you could write this rather meaningless computation:

```
My_horses[1].height = (My_horses[2].height + My_horses[3].height)/2;
```

I can think of no good reason why the height of one horse should be the average of two other horses' heights (unless there's some Frankenstein-like assembly going on) but it's a legal statement.

You can also use a complete structure element in an assignment statement:

```
My_horses[1] = My_horses[2];
```

This statement causes *all* the members of the structure `My_horses[2]` to be copied to the structure `My_horses[1]`, which means that the two structures become identical. The only other operation that's possible with a whole structure is to take its address using the `&` operator. You can't add, compare, or perform any other operations with a complete structure. To do those kinds of things, you must write your own functions.

Pointers to Structures

The ability to obtain the address of a structure raises the question of whether you can have pointers to a structure. Because you can take the address of a structure, the possibility of declaring a pointer to a structure does, indeed, naturally follow. You use the notation that you've already seen with other types of variables:

```
struct horse *phorse;
```

This declares a pointer, `phorse`, that can store the address of a structure of type `horse`.

You can now set `phorse` to have the value of the address of a particular structure, using exactly the same kind of statement that you've been using for other pointer types, for example

```
phorse = &My_horses[1];
```

Now `phorse` points to the structure `My_horses[1]`. You can immediately reference elements of this structure through your pointer. So if you want to display the name member of this structure, you could write this:

```
printf("\nThe name is %s.", (*phorse).name);
```

The parentheses around the dereferenced pointer are essential, because the precedence of the member selection operator (the period) is higher than that of the pointer-dereferencing operator *.

However, there's another way of doing this, and it's much more readable and intuitive. You could write the previous statement as follows:

```
printf("\nThe name is %s.", phorse->name );
```

So you don't need parentheses or an asterisk. You construct the operator -> from a minus sign immediately followed by the greater-than symbol. The operator is sometimes called the **pointer to member** operator for obvious reasons. This notation is almost invariably used in preference to the usual pointer-dereferencing notation you used at first, because it makes your programs so much easier to read.

Dynamic Memory Allocation for Structures

You have virtually all the tools you need to rewrite Program 11.2 with a much more economical use of memory. In the original version, you allocated the memory for an array of 50 `horse` structures, even when in practice you probably didn't need anything like that amount.

To create dynamically allocated memory for structures, the only tool that's missing is an array of pointers to structures, which is declared very easily, as you can see in this statement:

```
struct horse *phorse[50];
```

This statement declares an array of 50 pointers to structures of type `horse`. Only memory for the pointers has been allocated by this statement. You must still allocate the memory necessary to store the actual members of each structure that you need.

TRY IT OUT: USING POINTERS WITH STRUCTURES

You can see the dynamic allocation of memory for structures at work in the following example:

```
/* Program 11.3 Pointing out the horses */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>          /* For malloc() */

int main(void)
{
  struct horse              /* Structure declaration */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
  };

  struct horse *phorse[50]; /* pointer to structure array declaration */
  int hcount = 0;           /* Count of the number of horses          */
  char test = '\0';         /* Test value for ending input            */

  for(hcount = 0; hcount < 50 ; hcount++ )
  {
    printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                          hcount?"nother " : "" );
    scanf(" %c", &test );
    if(tolower(test) == 'n')
      break;

    /* allocate memory to hold a structure    */
    phorse[hcount] = (struct horse*) malloc(sizeof(struct horse));

    printf("\nEnter the name of the horse: " );
    scanf("%s", phorse[hcount]->name );  /* Read the horse's name */
```

```
    printf("\nHow old is %s? ", phorse[hcount]->name );
    scanf("%d", &phorse[hcount]->age );  /* Read the horse's age  */

    printf("\nHow high is %s ( in hands )? ", phorse[hcount]->name );
    scanf("%d", &phorse[hcount]->height ); /* Read the horse's height */

    printf("\nWho is %s's father? ", phorse[hcount]->name );
    scanf("%s", phorse[hcount]->father );  /* Get the father's name */

    printf("\nWho is %s's mother? ", phorse[hcount]->name );
    scanf("%s", phorse[hcount]->mother );  /* Get the mother's name */
  }

  /* Now tell them what we know. */
  for(int i = 0 ; i < hcount ; i++ )
  {
    printf("\n\n%s is %d years old, %d hands high,",
                phorse[i]->name, phorse[i]->age, phorse[i]->height);
    printf(" and has %s and %s as parents.",
                              phorse[i]->father, phorse[i]->mother);
    free(phorse[i]);
  }
  return 0;
}
```

The output should be exactly the same as that from Program 11.2, given the same input.

How It Works

This looks very similar to the previous version, but it operates rather differently. Initially, you don't have any memory allocated for any structures. The declaration

```
struct horse *phorse[50]; /* pointer to structure array declaration */
```

defines only 50 pointers to structures of type `horse`. You still have to find somewhere to put the structures to which these pointers are going to point:

```
phorse[hcount] = (struct horse*) malloc(sizeof(struct horse));
```

In this statement, you allocate the space for each structure as it's required. Let's have a quick reminder of how the `malloc()` function works. The `malloc()` function allocates the number of bytes specified by its argument and returns the address of the block of memory allocated as a pointer to type `void`. In this case, you use the `sizeof` operator to provide the value required.

It's very important to use `sizeof` when you need the number of bytes occupied by a structure. It doesn't necessarily correspond to the sum of the bytes occupied by each of its individual members, so you're likely to get it wrong if you try to work it out yourself.

Variables other than type `char` are often stored beginning at an address that's a multiple of 2 for 2-byte variables, a multiple of 4 for 4-byte variables, and so on. This is called **boundary alignment** and it has nothing to do with C but is a hardware requirement where it applies. Arranging variables to be stored in memory like this makes the transfer of data between the processor and memory faster. This arrangement can result in unused bytes occurring between member variables of different types, though, depending on their sequence. These have to be accounted for in the number of bytes allocated for a structure. Figure 11-2 presents an illustration of how this can occur.

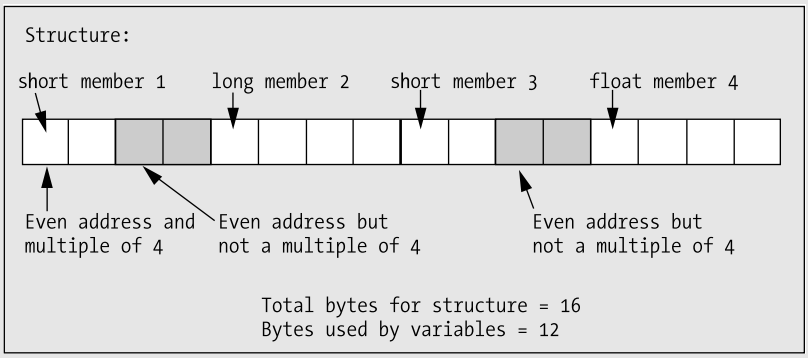

Figure 11-2. *The effect of boundary alignment on memory allocation*

As the value returned by `malloc()` is a pointer to `void`, you then cast this to the type you require with the expression `(struct horse*)`. This enables the pointer to be incremented or decremented correctly, if required.

```
scanf("%s", phorse[hcount]->name );  /* Read the horse's name     */
```

In this statement, you use the new notation for selecting members of a structure through a pointer. It's much clearer than `(*phorse[hcount]).name`. All subsequent references to members of a specific `horse` structure use this new notation.

Lastly in this program, you display a summary of all the data entered for each horse, freeing the memory as you go along.

More on Structure Members

So far, you've seen that any of the basic data types, including arrays, can be members of a structure. But there's more. You can also make a structure a member of a structure. Furthermore, not only can pointers be members of a structure, but also a pointer to a structure can be a member of a structure.

This opens up a whole new range of possibilities in programming with structures and, at the same time, increases the potential for confusion. Let's look at each of these possibilities in sequence and see what they have to offer. Maybe it won't be a can of worms after all.

Structures As Members of a Structure

At the start of this chapter, you examined the needs of horse breeders and, in particular, the necessity to manage a variety of details about each horse, including its name, height, date of birth, and so on. You then went on to look at Program 11.1, which carefully avoided date of birth and substituted age instead. This was partly because dates are messy things to deal with, as they're represented by three numbers and hold all the complications of leap years. However, you're now ready to tackle dates, using a structure that's a member of another structure.

You can define a structure type designed to hold dates. You can specify a suitable structure with the tag name `Date` with this statement:

```
struct Date
{
  int day;
  int month;
  int year;
};
```

Now you can define the structure `horse`, including a date-of-birth variable, like this:

```
struct horse
{
  struct Date dob;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

Now you have a single variable member within the structure that represents the date of birth of a horse, and this member is itself a structure. Next, you can define an instance of the structure `horse` with the usual statement:

```
struct horse Dobbin;
```

You can define the value of the member `height` with the same sort of statement that you've already seen:

```
Dobbin.height = 14;
```

If you want to set the date of birth in a series of assignment statements, you can use the logical extension of this notation:

```
Dobbin.dob.day = 5;
Dobbin.dob.month = 12;
Dobbin.dob.year = 1962;
```

You have a very old horse. The expression `Dobbin.dob.day` is referencing a variable of type `int`, so you can happily use it in arithmetic or comparative expressions. But if you use the expression `Dobbin.dob`, you would be referring to a `struct` variable of type `date`. Because this is clearly not a basic type but a structure, you can use it only in an assignment such as this:

```
Trigger.dob = Dobbin.dob;
```

This *could* mean that they're twins, but it doesn't guarantee it.

If you can find a good reason to do it, you can extend the notion of structures that are members of a structure to a structure that's a member of a structure that's a member of a structure. In fact, if you can make sense of it, you can continue with further levels of structure. Your C compiler is likely to provide for at least 15 levels of such convolution. But beware: if you reach this depth of structure nesting, you're likely to be in for a bout of repetitive strain injury just typing the references to members.

Declaring a Structure Within a Structure

You could declare the `Date` structure within the `horse` structure definition, as in the following code:

```
struct horse
{
  struct Date
  {
    int day;
    int month;
    int year;
  } dob;

  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

This has an interesting effect. Because the declaration is enclosed within the scope of the `horse` structure definition, it doesn't exist outside it, and so it becomes impossible to declare a `Date` variable external to the `horse` structure. Of course, each instance of a `horse` type variable would contain the `Date` type member, `dob`. But a statement such as this

```
struct date my_date;
```

would cause a compiler error. The message generated will say that the structure type `date` is undefined. If you need to use `date` outside the structure `horse`, its definition must be placed outside of the `horse` structure.

Pointers to Structures As Structure Members

Any pointer can be a member of a structure. This includes a pointer that points to a structure. A pointer structure member that points to the same type of structure is also permitted. For example, the `horse` type structure could contain a pointer to a `horse` type structure. Interesting, but is it of any use? Well, as it happens, yes.

TRY IT OUT: POINTERS TO STRUCTURES AS STRUCTURE MEMBERS

You can demonstrate a structure containing a pointer to a structure of the same type with a modification of the last example:

```
/* Program 11.4   Daisy chaining the horses */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main(void)
{
  struct horse                        /* Structure declaration       */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
```

```
  struct horse *next;              /* Pointer to next structure   */
};

struct horse *first = NULL;        /* Pointer to first horse      */
struct horse *current = NULL;      /* Pointer to current horse    */
struct horse *previous = NULL;     /* Pointer to previous horse   */

char test = '\0';                  /* Test value for ending input */

for( ; ; )
{
  printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                    first != NULL?"nother " : "" );
  scanf(" %c", &test );
  if(tolower(test) == 'n')
    break;

  /* Allocate memory for a structure */
  current = (struct horse*) malloc(sizeof(struct horse));

  if(first == NULL)
    first = current;               /* Set pointer to first horse  */

  if(previous != NULL)

    previous -> next = current; /* Set next pointer for previous horse */

  printf("\nEnter the name of the horse: ");
  scanf("%s", current -> name);    /* Read the horse's name       */

  printf("\nHow old is %s? ", current -> name);
  scanf("%d", &current -> age);    /* Read the horse's age        */

  printf("\nHow high is %s ( in hands )? ", current -> name );
  scanf("%d", &current -> height);  /* Read the horse's height    */

  printf("\nWho is %s's father? ", current -> name);
  scanf("%s", current -> father);   /* Get the father's name      */

  printf("\nWho is %s's mother? ", current -> name);
  scanf("%s", current -> mother);   /* Get the mother's name      */

  current->next = NULL;            /* In case it's the last...    */
  previous = current;              /* Save address of last horse  */
}

/* Now tell them what we know. */
current = first;                   /* Start at the beginning      */

while (current != NULL)      /* As long as we have a valid pointer */
{ /* Output the data*/
  printf("\n\n%s is %d years old, %d hands high,",
                    current->name, current->age, current->height);
```

```
 printf(" and has %s and %s as parents.", current->father,
                                          current->mother);
    previous = current;    /* Save the pointer so we can free memory */
    current = current->next;          /* Get the pointer to the next */
    free(previous);                   /* Free memory for the old one */
  }
  return 0;
}
```

This example should produce the same output as Program 11.3 (given the same input), but here you have yet another mode of operation.

How It Works

This time, not only do you have no space for structures allocated, but also you have only three pointers defined initially. These pointers are declared and initialized in these statements:

```
struct horse *first = NULL;          /* Pointer to first horse      */
struct horse *current = NULL;        /* Pointer to current horse    */
struct horse *previous = NULL;       /* Pointer to previous horse   */
```

Each of these pointers has been defined as a pointer to a `horse` structure. The pointer `first` is used solely to store the address of the first structure. The second and third pointers are working storage: `current` holds the address of the current `horse` structure that you're dealing with, and `previous` keeps track of the address of the previous structure that was processed.

You've added a member to the structure `horse` with the name `next`, which is a pointer to a `horse` type structure. This will be used to link together all the horses you have, where each `horse` structure will have a pointer containing the address of the next. The last structure will be an exception, of course: its `next` pointer will be `NULL`. The structure is otherwise exactly as you had previously. It's shown in Figure 11-3.

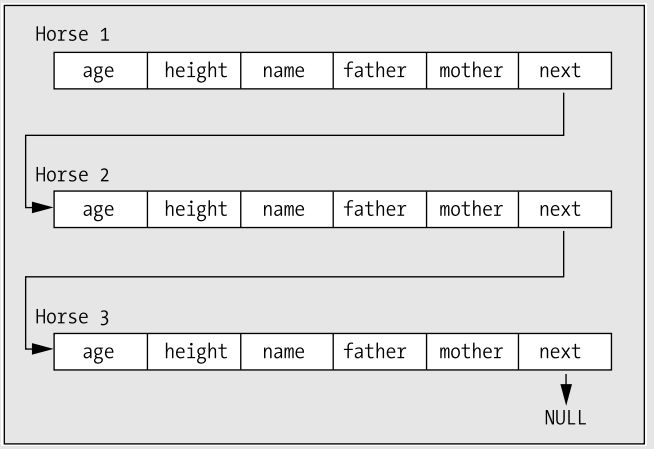

Figure 11-3. *A sequence of linked horses*

The input loop is the following:

```
for( ; ; )
{
  ...
}
```

The input loop is now an indefinite loop, because you don't have an array to worry about. You don't need to mess around with indexes. It's also unnecessary to keep count of how many sets of data are read in, so you don't need the variable `hcount` or the loop variable `i`. Because you allocate memory for each horse, you can just take them as they come.

The initial statements in the loop are the following:

```
printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                    first != NULL?"nother " : "" );
scanf(" %c", &test );
if(tolower(test) == 'n')
  break;
```

After the prompt, you exit from the loop if the response `'N'` or `'n'` is detected. Otherwise, you expect another set of structure members to be entered. You use the pointer `first` to get a slightly different prompt on the second and subsequent iterations, because the only time it will be `NULL` is on the first loop iteration.

Assuming you get past the initial question in the loop, you execute these statements:

```
current = (struct horse*) malloc(sizeof(struct horse));

if(first == NULL)
  first = current;                /* Set pointer to first horse  */

if(previous != NULL)
  previous -> next = current; /* Set next pointer for previous horse */
```

On each iteration, you allocate the memory necessary for the current structure. To keep things short, you don't check for a `NULL` return from `malloc()`, although you ought to do this in practice.

If the pointer `first` is `NULL`, you must be on the first loop iteration, and this must be the first structure about to be entered. Consequently, you set the pointer `first` to the pointer value that you've just obtained from `malloc()`, which is stored in the variable `current`. The address in `first` is the key to accessing the first horse in the chain. You can get to any of the others by starting with the address in `first` and then looking in the member pointer `next` to obtain the address of the next structure. You can step from there to the next structure and so on to any horse in the sequence.

The `next` pointer always needs to point to the next structure if there is one, but the address of the next structure can be determined only once you actually have the next structure. Therefore, on the second and subsequent iterations, you store the address of the current structure in the `next` member of the previous structure, whose address you'll have saved in `previous`. On the first iteration, the pointer `previous` will be `NULL` at this point, so of course you do nothing.

At the end of the loop, following all the input statements, you have these statements:

```
current->next = NULL;             /* In case it's the last...    */
previous = current;               /* Save address of last horse  */
```

The pointer `next` in the structure pointed to by `current`, which you're presently working with, is set to `NULL` in case this is the last structure and there's no next structure. If there is a next structure, this pointer `next` will be filled in on the next iteration. The pointer `previous` is set to `current` and is ready for the next iteration, when the `current` structure will indeed be the `previous` structure.

The strategy of the program is to generate a daisy chain of `horse` structures, in which the `next` member of each structure points to the next structure in the chain. The last is an exception because there's no next `horse`, so the `next` pointer contains `NULL`. This arrangement is called a **linked list**.

Once you have the `horse` data in a linked list, you process it by starting with the first structure and then getting the next structure through the pointer member `next`. When the pointer `next` is `NULL`, you know that you've reached the end of the list. This is how you generate the output list of all the input.

Linked lists are invaluable in applications in which you need to process an unknown number of structures, such as you have here. The main advantages of a linked list relate to memory usage and ease of handling. You occupy only the minimum memory necessary to store and process the list. Even though the memory used may be fragmented, you have no problem progressing from one structure to the next. As a consequence, in a practical situation in which you may need to deal with several different types of objects simultaneously, each can be handled using its own linked list, with the result that memory use is optimized. There is one small cloud associated with this—as there is with any silver lining—and it's that you pay a penalty in slower access to the data, particularly if you want to access it randomly.

The output process shows how a linked list is accessed as it steps through the linked list you've created with these statements:

```
current = first;                    /* Start at the beginning      */

while (current != NULL)      /* As long as we have a valid pointer */
{ /* Output the data*/
  printf("\n\n%s is %d years old, %d hands high,",
                    current->name, current->age, current->height);
  printf(" and has %s and %s as parents.", current->father,
                                          current->mother);
  previous = current;    /* Save the pointer so we can free memory */
  current = current->next;          /* Get the pointer to the next */
  free(previous);                   /* Free memory for the old one */
}
```

The `current` pointer controls the output loop, and it's set to `first` at the outset. Remember that the `first` pointer contains the address of the first structure in the list. The loop steps through the list, and as the members of each structure are displayed, the address stored in the member `next`, which points to the next structure, is assigned to `current`.

The memory for the structure displayed is then freed. It's obviously fairly essential that you only free the memory for a structure once you have no further need to reference it. It's easy to fall into the trap of putting the call of the function `free()` immediately after you've output all of the member values for the current structure. This would create some problems, because then you couldn't legally reference the current structure's `next` member to get the pointer to the next `horse` structure.

For the last structure in the linked list, the pointer `next` will contain `NULL` and the loop will terminate.

Doubly Linked Lists

A disadvantage of the linked list that you created in the previous example is that you can only go forward. However, a small modification of the idea gives you the **doubly linked list**, which will allow you to go through a list in either direction. The trick is to include an extra pointer in each structure to store the address of the previous structure in addition to the pointer to the next structure.

TRY IT OUT: DOUBLY LINKED LISTS

You can see a doubly linked list in action in a modified version of Program 11.4:

```
/* Program 11.5 Daisy chaining the horses both ways */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main(void)
{
  struct horse                        /* Structure declaration       */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
    struct horse *next;             /* Pointer to next structure     */
    struct horse *previous;         /* Pointer to previous structure */
  };

  struct horse *first = NULL;         /* Pointer to first horse      */
  struct horse *current = NULL;       /* Pointer to current horse    */
  struct horse *last = NULL;          /* Pointer to previous horse   */

  char test = '\0';                   /* Test value for ending input */

  for( ; ; )
  {
    printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                       first == NULL?"nother " : "");
    scanf(" %c", &test );
    if(tolower(test) == 'n')
      break;

    /* Allocate memory for each new horse structure */
    current = (struct horse*)malloc(sizeof(struct horse));

    if( first == NULL )
    {
      first = current;                /* Set pointer to first horse  */
     current->previous = NULL;
    }
    else
    {
      last->next = current;    /* Set next address for previous horse */
      current->previous = last; /* Previous address for current horse */
    }

    printf("\nEnter the name of the horse: ");
    scanf("%s", current -> name );    /* Read the horse's name   */
```

```
    printf("\nHow old is %s? ", current -> name);
    scanf("%d", &current -> age);     /* Read the horse's age    */

    printf("\nHow high is %s ( in hands )? ", current -> name);
    scanf("%d", &current -> height);  /* Read the horse's height */

    printf("\nWho is %s's father? ", current -> name);
    scanf("%s", current -> father);   /* Get the father's name   */

    printf("\nWho is %s's mother? ", current -> name);
    scanf("%s", current -> mother);   /* Get the mother's name   */

    current -> next = NULL;      /* In case it's the last horse..*/
    last = current;              /* Save address of last horse   */
  }

  /* Now tell them what we know. */
  while(current != NULL)         /* Output horse data in reverse order */
  {
    printf("\n\n%s is %d years old, %d hands high,",
              current->name, current->age, current->height);
    printf(" and has %s and %s as parents.", current->father,
                                             current->mother);
    last = current;       /* Save pointer to enable memory to be freed */
    current = current->previous; /* current points to previous in list */
    free(last);                 /* Free memory for the horse we output */
  }
  return 0;
}
```

For the same input, this program should produce the same output as before, except that the data on horses entered is displayed in reverse order to that of entry—just to show that you can do it.

How It Works

The initial pointer declarations are now as follows:

```
  struct horse *first = NULL;         /* Pointer to first horse      */
  struct horse *current = NULL;       /* Pointer to current horse    */
  struct horse *last = NULL;          /* Pointer to previous horse   */
```

You change the name of the pointer recording the `horse` structure entered on the previous iteration of the loop to `last`. This name change isn't strictly necessary, but it does help to avoid confusion with the structure member `previous`.

The structure `horse` is declared as follows:

```
  struct horse                        /* Structure declaration       */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
```

```
    struct horse *next;             /* Pointer to next structure     */
    struct horse *previous;         /* Pointer to previous structure */
  };
```

The `horse` structure now contains two pointers: one to point forward in the list, called `next`, the other to point backward to the preceding structure, called `previous`. This allows the list to be traversed in either direction, as you demonstrate by the fact that you output the data at the end of the program in reverse order.

Aside from the output, the only changes to the program are to add the statements that take care of the entries for the pointer structure member `previous`. In the beginning of the input loop you have the following:

```
    if( first == NULL )
    {
      first = current;               /* Set pointer to first horse   */
      current->previous = NULL;
    }
    else
    {
      last->next = current;    /* Set next address for previous horse */
      current->previous = last; /* Previous address for current horse */
    }
```

Here, you take the option of writing an `if` with an `else`, rather than the two `if`s you had in the previous version. The only material difference is setting the value of the structure member `previous`. For the first structure, `previous` is set to `NULL`, and for all subsequent structures it's set to the pointer `last`, whose value was saved on the preceding iteration.

The other change is at the end of the input loop:

```
last = current;                         /* Save address of last horse  */
```

This statement is added to allow the pointer `previous` in the next structure to be set to the appropriate value, which is the `current` structure that you're recording in the variable `last`.

The output process is virtually the same as in the previous example, except that you start from the last structure in the list and work back to the first.

Bit-Fields in a Structure

Bit-fields provide a mechanism that allows you to define variables that are each one or more binary bits within a single integer word, which you can nevertheless refer to explicitly with an individual member name for each one.

■ **Note** Bit-fields are used most frequently when memory is at a premium and you're in a situation in which you must use it as sparingly as possible. This is rarely the case these days so you won't see them very often. Bit-fields will slow your program down appreciably compared to using standard variable types. You must therefore assess each situation upon its merits to decide whether the memory savings offered by bit-fields are worth this price in execution speed for your programs. In most instances, bit-fields won't be necessary or even desirable, but you need to know about them.

An example of declaring a bit-field is shown here:

```
struct
{
  unsigned int flag1 : 1;
  unsigned int flag2 : 1;
  unsigned int flag3 : 2;
  unsigned int flag4 : 3;
} indicators;
```

This defines a variable with the name `indicators` that's an instance of an anonymous structure containing four bit-fields with the names `flag1` through `flag4`. These will all be stored in a single word, as illustrated in Figure 11-4.

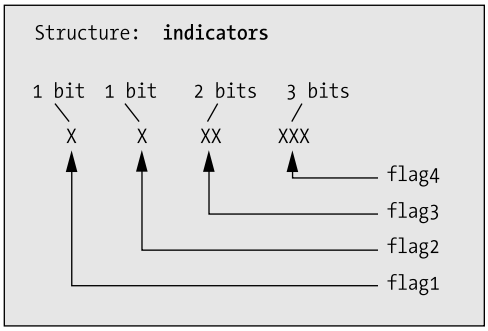

Figure 11-4. *Bit-fields in a structure*

The first two bit-fields, being a single bit specified by the 1 in their definition, can only assume the values 0 or 1. The third bit-field, `flag3`, has 2 bits and so it can have a value from 0 to 3. The last bit-field, `flag4`, can have values from 0 to 7, because it has 3 bits. These bit-fields are referenced in the same manner as other structure members, for example

```
indicators.flag4 = 5;
indicators.flag3 = indicators.flag1 = 1;
```

You'll rarely, if ever, have any need for this facility. I've included bit-fields here for the sake of completeness and for that strange off chance that one day bit-fields will be just what you need in a particularly tight memory situation.

Structures and Functions

Because structures represent such a powerful feature of the C language, their use with functions is very important. You'll now look at how you can pass structures as arguments to a function and how you can return a structure from a function.

Structures As Arguments to Functions

There's nothing unusual in the method for passing a structure as an argument to a function. It's exactly the same as passing any other variable. Analogous to the `horse` structure, you could create this structure:

```
struct family
{
  char name[20];
  int age;
  char father[20];
  char mother[20];
};
```

You could then construct a function to test whether two members of the type `family` are siblings:

```
bool siblings(struct family member1, struct family member2)
{
  if(strcmp(member1.mother, member2.mother) == 0)
    return true;
  else
    return false;
}
```

This function has two arguments, each of which is a structure. It simply compares the strings corresponding to the member `mother` for each structure. If they're the same, they are siblings and 1 is returned. Otherwise, they can't be siblings so 0 is returned. You're ignoring the effects of divorce, in vitro fertilization, cloning, and any other possibilities that may make this test inadequate.

Pointers to Structures As Function Arguments

Remember that a copy of the value of an argument is transferred to a function when it's called. If the argument is a large structure, it can take quite a bit of time, as well as occupying whatever additional memory the copy of the structure requires. Under these circumstances, you should use a pointer to a structure as an argument. This avoids the memory consumption and the copying time, because now only a copy of the pointer is made. The function will access the original structure directly through the pointer. More often than not, structures are passed to a function using a pointer, just for these reasons of efficiency. You could rewrite the `siblings()` function like this:

```
bool siblings(struct family *member1, struct family *member2)
{
  if(strcmp(member1->mother, member2->mother) == 0)
    return true;
  else
    return false;
}
```

Now, there is a downside to this. The pass-by-value mechanism provides good protection against accidental modification of values from within a called function. You lose this if you pass a pointer to a function. On the upside, if you don't need to modify the values pointed to by a pointer argument (you just want to access and use them, for instance), there's a technique for getting a degree of protection, even though you're passing pointers to a function.

Have another look at the last `siblings()` function. It doesn't need to modify the structures passed to it—in fact, it only needs to compare members. You could therefore rewrite it like this:

```
bool siblings(struct family const *pmember1, struct family const *pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

You'll recall the const modifier from earlier in the book, where you used it to make a variable effectively a constant. This function declaration specifies the parameters as type "pointer to constant family structure." This implies that the structures pointed to by the pointers transferred to the function will be treated as constants within the function. Any attempt to change those structures will cause an error message during compilation. Of course, this doesn't affect their status as variables in the calling program, because the const keyword applies only to the values while the function is executing.

Note the difference between the previous definition of the function and this one:

```
bool siblings(struct family *const pmember1, struct family *const pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

The indirection operator in each parameter definition is now in front of the keyword const, rather than in front of the pointer name as it was before. Does this make a difference? You bet it does. The parameters here are "constant pointers to structures of type family," not "pointers to constant structures." Now you're free to alter the structures themselves in the function, but you must not modify the addresses stored in the pointers. It's the pointers that are protected here, not the structures to which they point.

A Structure As a Function Return Value

There's nothing unusual about returning a structure from a function either. The function prototype merely has to indicate this return value in the normal way, for example:

```
struct horse my_fun(void);
```

This is a prototype for a function taking no arguments that returns a structure of type horse.

Although you can return a structure from a function like this, it's often more convenient to return a pointer to a structure. Let's explore this in more detail through a working example.

TRY IT OUT: RETURNING A POINTER TO A STRUCTURE

To demonstrate how returning a pointer to a structure works, you can rewrite the previous horse example in terms of humans and perform the input in a separate function:

```
/* Program 11.6 Basics of a family tree */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

struct Family *get_person(void);       /* Prototype for input function */

struct Date
{
  int day;
   int month;
  int year;
};
```

```
struct Family                          /* Family structure declaration */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;                /* Pointer to next structure    */
  struct Family *previous;           /* Pointer to previous structure */
};

int main(void)
{
  struct Family *first = NULL;        /* Pointer to first person      */
  struct Family *current = NULL;      /* Pointer to current person    */
  struct Family *last = NULL;         /* Pointer to previous person   */
  char more = '\0';                   /* Test value for ending input  */

  for( ; ; )
  {
    printf("\nDo you want to enter details of a%s person (Y or N)? ",
                                       first != NULL?"nother" : "");
    scanf(" %c", &more);
    if(tolower(more) == 'n')
      break;

    current = get_person();

    if(first == NULL)
    {
      first = current;                /* Set pointer to first Family  */
      last = current;                 /* Remember for next iteration  */
    }
    else
    {
      last->next = current;   /* Set next address for previous Family */
      current->previous = last;   /* Set previous address for current */
      last = current;             /* Remember for next iteration      */
    }
  }

  /* Now tell them what we know */

  /* Output Family data in reverse order */
  while (current  != NULL)
  {
    printf("\n%s was born %d/%d/%d, and has %s and %s as parents.",
              current->name, current->dob.day, current->dob.month,
              current->dob. year, current->father,  current->mother );

    last = current;      /* Save pointer to enable memory to be freed */
    current = current->previous;   /* current points to previous list */
    free(last);          /* Free memory for the Family we output      */
  }
```

```
 return 0;
}

/*   Function to input data on Family members   */
struct Family *get_person(void)
{
  struct Family *temp;          /* Define temporary structure pointer */

  /* Allocate memory for a structure */
  temp = (struct Family*) malloc(sizeof(struct Family));

  printf("\nEnter the name of the person: ");
  scanf("%s", temp -> name );         /* Read the Family's name       */

  printf("\nEnter %s's date of birth (day month year); ", temp->name);
  scanf("%d %d %d", &temp->dob.day, &temp->dob.month, &temp->dob.year);

  printf("\nWho is %s's father? ", temp->name );
  scanf("%s", temp->father );         /* Get the father's name        */

  printf("\nWho is %s's mother? ", temp->name );
  scanf("%s", temp -> mother );       /* Get the mother's name        */

  temp->next = temp->previous = NULL; /* Set pointers to NULL         */

  return temp;                  /* Return address of Family structure */
}
```

How It Works

Although this looks like a lot of code, you should find this example quite straightforward. It operates similarly to the previous example, but it's organized as two functions instead of one.

The first structure declaration is the following:

```
struct Date
{
  int day;
  int month;
  int year;
};
```

This defines a structure type `Date` with three members, `day`, `month`, and `year`, which are all declared as integers. No instances of the structure are declared at this point. The definition precedes all the functions in the source file so it is accessible from within any function that appears subsequently in the file.

The next structure declaration is the following:

```
struct Family                         /* Family structure declaration  */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;                /* Pointer to next structure    */
  struct Family *previous;           /* Pointer to previous structure */
};
```

This defines a structure type `Family`, which has a `Date` type structure as its first member. It then has three conventional `char` arrays as members. The last two members are pointers to structures. They're intended to allow a doubly linked list to be constructed, being pointers to the next and previous structures in the list, respectively.

The fact that both structure declarations are external to all the functions and are therefore available globally is an important difference between this and the previous examples. This is necessary here because you want to define `Family` structure variables in both the functions `main()` and `get_person()`.

■ **Note** Only the specification of the structure type is accessible globally. All the variables of type `Family` declared within each function are local in scope to the function in which they're declared.

The function `get_person()` has this prototype:

```
struct Family *get_person(void);  /* Prototype for input function   */
```

This indicates that the function accepts no arguments but returns a pointer to a `Family` structure.

The process parallels the operation of Program 11.5, with the differences that you have global structure type declarations and you input a structure within a separate function.

After verifying that the user wants to enter data by checking his or her response in `more`, the function `main()` calls the function `get_person()`. Within the function `get_person()`, you declare this pointer:

```
struct Family *temp;          /* Define temporary structure pointer */
```

This is a "pointer to a `Family` type structure" and it has local scope. The fact that the declaration of the structure type is global has no bearing on the scope of actual instances of the structure. The scope of each instance that you declare will depend on where the declaration is placed in the program.

The first action within the function `get_person()` is the following:

```
temp = (struct Family*) malloc(sizeof(struct Family));
```

This call to `malloc()` obtains sufficient memory to store a structure of type `Family` and stores the address that's returned in the pointer variable, `temp`. Although `temp` is local and will go out of scope at the end of the function `get_person()`, the memory allocated by `malloc()` is more permanent. It remains until you free it yourself within the program somewhere, or until you exit from the program completely.

The function `get_person()` reads in all the basic data for a person and stores that data in the structure pointed to by `temp`. As it stands, the function will accept any values for the date, but in a real situation you would include code for data validity checking. You could verify that the month value is from 1 to 12 and the day value is valid for the month entered. Because a birth date is being entered, you might verify that it isn't in the future.

The last statement in the function `get_person()` is the following:

```
return temp;                  /* Return address of Family structure */
```

This returns a copy of the pointer to the structure that it has created. Even though `temp` will no longer exist after the return, the address that it contains that points to the memory block obtained from `malloc()` is still valid.

Back in `main()`, the pointer that's returned is stored in the variable `current` and is also saved in the variable `first` if this is the first iteration. You do this because you don't want to lose track of the first structure in the list. You also save the pointer `current` in the variable `last`, so that on the next iteration you can fill in the backward pointer member, `previous`, for the current person whose data you've just obtained.

After all the input data has been read, the program outputs a summary to the screen in reverse order, in a similar fashion to the previous examples.

An Exercise in Program Modification

Perhaps we ought to produce an example combining both the use of pointers to structures as arguments and the use of pointers to structures as return values. You can declare some additional pointers, p_to_pa and p_to_ma, in the structure type Family in the previous example, Program 11.6, by changing that structure declaration as follows:

```
struct Family                          /* Family structure declaration  */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;                 /* Pointer to next structure     */
  struct Family *previous;             /* Pointer to previous structure */
  struct Family *p_to_pa;              /* Pointer to father structure   */
  struct Family *p_to_ma;              /* Pointer to mother structure   */
};
```

Now you can record the addresses of related structures in the pointer members p_to_pa and p_to_ma. You'll need to set them to NULL in the get_person() function by adding the following statement just before the return statement:

```
  temp->p_to_pa = temp->p_to_ma = NULL;       /* Set pointers to NULL  */
```

You can now augment the program with some additional functions that will fill in your new pointers p_to_pa and p_to_ma once data for everybody has been entered. You could code this by adding two functions. The first function, set_ancestry(), will accept pointers to Family structures as arguments and check whether the structure pointed to by the second argument is the father or mother of the structure pointed to by the first argument. If it is, the appropriate pointer will be updated to reflect this, and true will be returned; otherwise false will be returned. Here's the code:

```
bool set_ancestry(struct Family *pmember1, struct Family *pmember2)
{
  if(strcmp(pmember1->father, pmember2->name) == 0)
  {
    pmember1->p_to_pa = pmember2;
    return true;
  }

  if( strcmp(pmember1->mother, pmember2->name) == 0)
  {
    pmember1->p_to_ma = pmember2;
    return true;
  }
  else
    return false;
}
```

The second function will test all possible relationships between two Family structures:

```
/* Fill in pointers for mother or father relationships */
bool related (struct Family *pmember1, struct Family *pmember2)
{
  return set_ancestry(pmember1, pmember2) ||
                           set_ancestry(pmember2, pmember1);
}
```