Chapter 11: Structures and Unions

WHAT IS A STRUCTURE ?

A structure is a user defined data type. Using a structure you have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a combination of several different previously defined data types, including other structures we have defined. A simple definition is, "a structure is a grouping of related data in a way convenient to the programmer or user of the program." The best way to understand a structure is to look at an example, so if you will load and display struct1.c, we will do just that.

The program begins with a structure definition. The keyword **struct** is followed by three simple variables between the braces, which are the components of the structure. After the closing brace, you will find two variable names listed, **boy**, and **girl**. According to the definition of a structure, **boy** is now a variable composed of three elements, **initial**, **age**, and **grade**. Each of the three fields are associated with **boy**, and each can store a variable of its respective type. The variable named **girl** is also a variable containing three fields with the same names as those of **boy** but are actually different variables. We have therefore defined 6 simple variables, but they are grouped into 2 variables of a structure type.

A SINGLE COMPOUND VARIABLE

Lets examine the variable named **boy** more closely. As stated above, each of the three elements of **boy** are simple variables and can be used anywhere in a C program where a variable of their type can be used. For example, the **age** element is an integer variable and can therefore be used anywhere in a C program where it is legal to use an integer variable, in calculations, as a counter, in I/O operations, etc. We now have the problem of defining how to use the simple variable named **age** which is a part of the compound variable named **boy**. To do so we use both names with a decimal point between them with the major name first. Thus **boy.age** is the complete variable name for the **age** field of **boy**. This construct can be used anywhere in a C program that it is desired to refer to this field. In fact, it is illegal to use the name **boy** or **age** alone because they are only partial definitions of the complete field. Alone, the names refer to nothing. (Actually the name **boy** alone does have meaning when used with a modern C compiler. We will discuss this later.)

ASSIGNING VALUES TO THE VARIABLES

Using the above definition, we can assign a value to each of the three fields of **boy** and each of the three fields of **girl**. Note carefully that **boy.initial** is actually a char type variable, because it was defined as one in the structure, so it must be assigned a character of data. In line 12, **boy.initial** is assigned the character R in agreement with the above rules. The remaining two fields of **boy** are assigned values in accordance with their respective types. Finally the three fields of **girl** are assigned values but in a different order to illustrate that the order of assignment is not critical. You will notice that we used the value of the boy's age when we defined the girl's age. This illustrates the use of one member of the structure. Figure 11-1 is a graphical

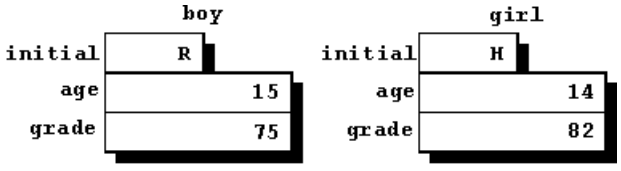

Figure 11-1

representation of the data following execution of line 18.

HOW DO WE USE THE RESULTING DATA ?

Now that we have assigned values to the six simple variables, we can do anything we desire with them. In order to keep this first example simple, we will simply print out the values to see if they really do exist as assigned. If you carefully inspect the **printf()** statements, you will see that there is nothing special about them. The compound name of each variable is specified because that is the only valid name by which we can refer to these variables.

Structures are a very useful method of grouping data together in order to make a program easier to write and understand. This first example is too simple to give you even a hint of the value of using structures, but continue on through these lessons and eventually you will see the value of using structures. Compile and run struct1.c and observe the output.

AN ARRAY OF STRUCTURES

Load and display the next program named struct2.c. This program contains the same structure definition as before but this time we define an array of 12 variables named **kids**. It should be clear that this program contains 12 times 3 = 36 simple variables, each of which can store one item of data provided that it is of the correct type. We also define a simple variable named **index** for use in the for loops.

In order to assign each of the fields a value, we use a for loop and each pass through the loop results in assigning a value to each of the fields of one structure variable. One pass through the loop assigns all of the values for one of the kids. This would not be a very useful way to assign data in a real situation, but a loop could read the data in from a file and store it in the correct fields in a real application. You might consider this the crude beginning of a data base, which it is.

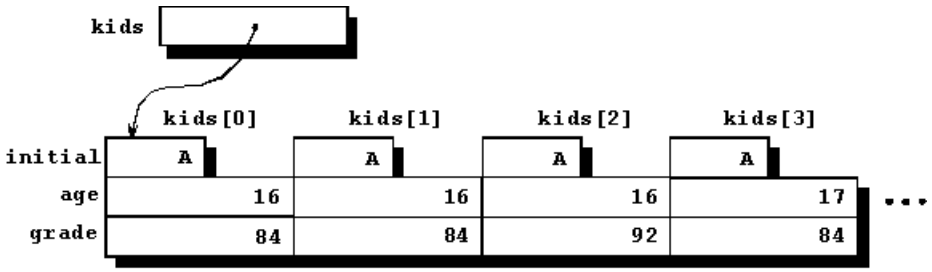

Figure 11-2

In the next few instructions of this program we assign new values to some of the fields to illustrate the method used to accomplish this. It should be self explanatory, so no additional comments will be given. Figure 11-2 is a graphical representation of the data for this program following execution of line 22.

A RECENT UPGRADE TO THE C LANGUAGE

All good C compilers will allow you to copy an entire structure with one statement. This is a fairly recent addition to the C language and is a part of the ANSI standard, so you should feel free to use it with your C compiler if it is available. Line 24 is an example of using a structure assignment. In this statement, all 3 fields of **kids[4]** are copied into their respective fields of **kids[10]**.

WE FINALLY DISPLAY ALL OF THE RESULTS

The last few statements contain a for loop in which all of the generated values are displayed in a formatted list. Compile and run the program to see if it does what you expect it to do. You will need to remove line 24 if your compiler does not support structure assignments.

USING POINTERS AND STRUCTURES TOGETHER

Examine the file named struct3.c for an example of using pointers with structures. This program is identical to the last program except that it uses pointers for some of the operations.

The first difference shows up in the definition of variables following the structure definition. In this program we define a pointer named **point** which is defined as a pointer that points to the structure. It would be illegal to try to use this pointer to point to any other variable type. There is a very definite reason for this restriction in C as we have alluded to earlier and will review in the next few paragraphs.

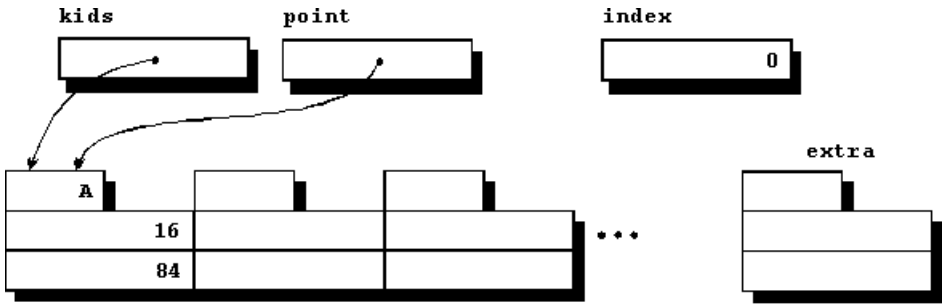

Figure 11-3

The next difference is in the for loop where we use the pointer for accessing the data fields. Recall from chapter 8 of this tutorial that we said that the name of an array is actually a pointer to the first element of the array. Since **kids** is a pointer variable that points to the first element of the array which is a structure, we can define **point** in terms of **kids**. The variable **kids** is a constant so it cannot be changed in value, but **point** is a pointer variable and can be assigned any value consistent with its being required to point to the structure. If we assign the value of **kids** to **point** then it should be clear that **point** will also point to the first element of the array, a structure containing three fields. Figure 11-3 is a graphical representation of the data space following the first pass through the loop starting in line 14.

POINTER ARITHMETIC

Adding 1 to **point** will now cause it to point to the second field of the array because of the way pointers are handled in C. The system knows that the structure contains three variables and it knows how many memory elements are required to store the complete structure. Therefore if we tell it to add one to the pointer, it will actually add the number of memory elements required to get to the next element of the array. If, for example, we were to add 4 to the pointer, it would advance the value of the pointer 4 times the size of the structure, resulting in it pointing 4 elements farther along the array. This is the reason a pointer cannot be used to point to any data type other than the one for which it was defined.

Now to return to the program displayed on your monitor. It should be clear from the previous discussion that as we go through the loop, the pointer will point to the beginning of one of the array elements each time. We can therefore use the pointer to reference the various elements of each of the structures as we go through the loop. Referring to the elements of a structure with a pointer occurs so often in C that a special method of doing that was devised. Using `point->initial` is the same as

using `(*point).initial` which is really the way we did it in the last two programs. Remember that ***point** is the stored data to which the pointer points and the construct should be clear. The "->" is made up of the minus sign and the greater than sign. You will find experienced C programmers using this pointer dereference profusely when you read their code in magazines and other publications.

Since the pointer points to the structure, we must once again define which of the elements we wish to refer to each time we use one of the elements of the structure. There are, as we have seen, several different methods of referring to the members of the structure. When executing the for loop used for output at the end of the program, we use three different methods of referring to the structure elements. This would be considered very poor programming practice, but is done this way here to illustrate to you that they all lead to the same result. This program will probably require some study on your part to fully understand, but it will be worth your time and effort to grasp these principles.

Lines 30 and 31 are two additional examples of structure assignment which do nothing useful, but are included here for your benefit. Compile and run this program, and once again, if your compiler does not support structure assignment, you will need to remove lines 30 and 31.

NESTED AND NAMED STRUCTURES

Examine the file named nested.c for an example of a nested structure. The structures we have seen so far have been very simple, although useful. It is possible to define structures containing dozens and even hundreds or thousands of elements but it would be to the programmers advantage not to define all of the elements at one pass but rather to use a hierarchical structure definition. This will be illustrated with the program on your monitor.

The first structure contains three elements but is followed by no variable name. We therefore have not defined any variables, only a structure, but since we have included a name at the beginning of the structure, the structure is named **person**. The name **person** can be used to refer to the structure but not to any variable of this structure type. It is therefore a new type that we have defined, and we can use the new type in the same way we use **int**, **char**, or any other types that exist in C. The only restriction is that this new name must always be associated with the keyword **struct**.

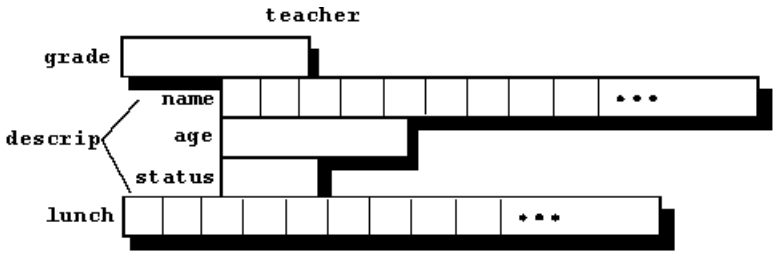

Figure 11-4

The next structure definition contains three fields with the middle field being the previously defined structure which we named **person**. The variable which has the type of **person** is named **descrip**. So the new structure contains two simple variables, **grade** and a string named **lunch**, and the structure named **descrip**. Since **descrip** contains three variables, the new structure actually contains 5 variables. This structure is also given a name **alldat**, which is another type definition. Finally we define an array of 53 variables each with the structure defined by the type **alldat**, and each with the name **student**. If that is clear, you will see that we have defined a total of 53 times 5 variables, each of which is capable of storing a value.

Since we have a new type definition we can use it to define two more variables. The variables

teacher and **sub** are defined in line 20 to be variables of the type **alldat**, so that each of these two variables contain 5 fields in which we can store data. Figure 11-4 is a graphical representation of the variable named **teacher** after it is defined in line 20.

NOW TO USE SOME OF THE FIELDS

In lines 22 through 26 of the program, we will assign values to each of the fields of **teacher**. The first field is the **grade** field and is handled just like the other structures we have studied because it is not part of the nested structure. Next we wish to assign a value to her **age** which is part of the nested structure. To address this field we start with the variable name **teacher** to which we append the name of the group **descrip**, and then we must define which field of the nested structure we are interested in, so we append the variable name **age**. The teachers **status** is handled in exactly the same manner as her **age**, but the last two fields are assigned strings using the string copy function **strcpy()** which must be used for string assignment. Notice that the variable names in the **strcpy()** function are still variable names even though they are made up of several parts each.

The variable **sub** is assigned nonsense values in much the same way, but in a different order since they do not have to occur in any required order. Finally, a few of the student variables are assigned values for illustrative purposes and the program ends. None of the values are printed for illustration since several were printed in the last examples.

Compile and run this program, but when you run it, you may get a stack overflow error. C uses its own internal stack to store the automatic variables, but some C compilers use only a 2048 byte stack as a default. This program requires more than that for the defined structures so it will be necessary for you to increase the stack size. Consult your compiler documentation for details concerning the method of increasing the stack size. There is no standard way to do this. There is another way around this problem, and that is to move the variable definitions outside of the main program where they will be external variables and therefore static. The result is that they will not be kept on the internal stack and the stack will not overflow. It would be good experience for you to try both methods of fixing this problem.

MORE ABOUT STRUCTURES

It is possible to continue nesting structures until you get totally confused. If you define them properly, the computer will not get confused because there is no stated limit as to how many levels of nesting are allowed. There is probably a practical limit of three beyond which you will probably get confused, but the language has no limit. In addition to nesting, you can include as many structures as you desire in any level of structures, such as defining another structure prior to **alldat** and using it in **alldat** in addition to using **person**. The structure named **person** could be included in **alldat** two or more times if desired, as could pointers to it.

Structures can contain arrays of other structures which in turn can contain arrays of simple types or other structures. It can go on and on until you lose all reason to continue. I am only trying to illustrate to you that structures are very valuable and you will find them great aids to programming if you use them wisely. Be conservative at first, and get bolder as you gain experience. Keep in mind that a structure is designed to group related data together.

More complex structures will not be illustrated here, but you will find examples of additional structures in the example programs included in the last chapter of this tutorial. For example, see the include file named vc.h on the distribution disk.

WHAT ARE UNIONS ?

Examine the file named union1.c for an example of a union. Simply stated, a union allows you a way

to look at the same data with different types, or to use the same data with different names.

In this example we have two elements to the union, the first part being the integer named **value**, which is stored as a two byte variable somewhere in the computers memory. The second element is made up of two character variables named **first** and **second**. These two variables are stored in the same storage locations that value is stored in, because that is what a union does. A union allows you to store different types of data in the same physical storage locations. In this case, you could put an integer number in **value**, then retrieve it in its two halves by getting each half using the two names **first** and **second**. This technique is often used to pack data bytes together when you are, for example, combining bytes to be used in the registers of the microprocessor.

Accessing the fields of the union are very similar to accessing the fields of a structure and will be left to you to determine by studying the example.

One additional note must be given here about the program. When it is run using some C compilers, the data will be displayed with two leading f's due to the hexadecimal output promoting the **char** type variables to **int** and extending the sign bit to the left. Converting the **char** type data fields to **int** type fields prior to display should remove the leading f's from your display. This will involve defining two new **int** type variables and assigning the **char** type variables to them. This will be left as an exercise for you. Note that the same problem will come up in a few of the later files in this tutorial.

Compile and run this program and observe that the data is read out as an **int** and as two **char** variables. The **char** variables may be reversed in order because of the way an **int** variable is stored internally in your computer. If your system reverses these variables, don't worry about it. It is not a problem but it can be a very interesting area of study if you are so inclined.

ANOTHER UNION EXAMPLE

Examine the file named union2.c for another example of a union, one which is much more common. Suppose you wished to build a large database including information on many types of vehicles. It would be silly to include the number of propellers on a car, or the number of tires on a boat. In order to keep all pertinent data, however, you would need those data points for their proper types of vehicles. In order to build an efficient data base, you would need several different types of data for each vehicle, some of which would be common, and some of which would be different. That is exactly what we are doing in the example program on your monitor.

In this program, we will define a complete structure, then decide which of the various types can go into it. We will start at the top and work our way down. First, we define a few constants with the #defines, and begin the program itself. We define a structure named **automobile** containing several fields which you should have no trouble recognizing, but we define no variables at this time.

A NEW CONCEPT, THE TYPEDEF

Next we define a new type of data with a **typedef**. This defines a complete new type that can be used in the same way that **int** or **char** can be used. Notice that the structure has no name, but at the end where there would normally be a variable name there is the name BOATDEF. We now have a new type, BOATDEF, that can be used to define a structure anyplace we would like to. Notice that this does not define any variables, only a new type. Using all caps for the name is a personal preference only and is not a C standard but is used by most experienced C programmers. It makes the **typedef** look different from a variable name.

We finally come to the big structure that defines our data using the building blocks already defined above. The structure is composed of 5 parts, two simple variables named **vehicle** and **weight**,

followed by the union, and finally the last two simple variables named **value** and **owner**. Of course the union is what we need to look at carefully here, so focus on it for the moment. You will notice that it is composed of four parts, the first part being the variable **car** which is a structure that we defined previously. The second part is a variable named **boat** which is a structure of the type BOATDEF previously defined. The third part of the union is the variable **airplane** which is a structure defined in place in the union. Finally we come to the last part of the union, the variable named **ship** which is another structure of the type BOATDEF.

I hope it is obvious to you that all four could have been defined in any of the three ways shown, but the three different methods were used to show you that any could be used. In practice, the clearest definition would probably have occurred by using the typedef for each of the parts.

WHAT DO WE HAVE NOW ?

We now have a structure that can be used to store any of four different kinds of data structures. The size of every record will be the size of that record containing the largest union. In this case part 1 is the largest union because it is composed of three integers, the others being composed of an integer and a character each. The first member of this union would therefore determine the size of all structures of this type. The resulting structure can be used to store any of the four types of data, but it is up to the programmer to keep track of what is stored in each variable of this type. The variable named **vehicle** was designed into this structure to keep track of the type of vehicle stored here. The four defines at the top of the page were designed to be used as indicators stored in the variable named **vehicle**.

A few examples of how to use the resulting structure are given in the next few lines of the program. Some of the variables are defined and a few of them are printed out for illustrative purposes.

The union is not used too frequently, and almost never by beginning programmers. You will encounter it occasionally so it is worth your effort to at least know what it is. You do not need to know the details of it at this time, so don't spend too much time studying it. When you do have a need for a variant structure, a union, you can learn it at that time. For your own benefit, however, do not slight the structure. You should use the structure often.

WHAT IS A BITFIELD ?

Load and display the program named bitfield.c for an example of how to define and use a bitfield. In this program, we have a union made up of a single **int** type variable in line 5 and the structure defined in lines 6 through 10. The structure is composed of three bitfields named **x**, **y**, and **z**. The variable named **x** is only one bit wide, the variable **y** is two bits wide and adjacent to the variable **x**, and the variable **z** is two bits wide and adjacent to **y**. Moreover, because the union causes the bits to be stored in the same memory location as the variable **index**, the variable **x** is the least significant bit of the variable **index**, **y** is the next two bits, and **z** is stored in the next two bits of **index**.

Compile and run the program and you will see that as the variable **index** is incremented by one each time through the loop, and you will see the bitfields of the union counting due to their respective locations within the int definition.

One thing must be pointed out, the bitfields must be defined as parts of an **unsigned int** or your compiler will issue an error message.

WHAT IS THE BITFIELD GOOD FOR ?

The bitfield is very useful if you have a lot of data to separate into individual bits or groups of bits. Many systems use some sort of a packed format to get lots of data stored in a few bytes. Your

imagination is your only limitation to the efficient use of this feature of C.

MORE STYLE ISSUES

Examine the file named style3.h for our first example of a header file that really looks like one. You will notice several constant definitions, a few structure definitions, and some prototypes. Nothing in this file generates anything that uses memory, since there are no variables defined and no code is defined here. Each of those use some memory, but all of the constructs in this file do nothing but create definitions which are then used by other portions of the program. This header file, if it is general enough, can be used by many different implementations.

Spend a few minutes and observe the style. Take notice especially of the order of the various entities. The constants are defined first, followed by the structures since they generally use one or more of the constants. Finally, the prototypes are defined since they often make use of one or more of the structures in their parameter lists or their return values.

Examine the file named style3.c which uses some of the definitions in the header style3.h header file. The observant student will notice that not everything that is defined in the header file is used in this implementation file, and it really doesn't need to be. Since a header file is meant to be general purpose, all things within the file will not be used every time the file itself is used in a program.

In this case, the structure named **alldat** is used in lines 14 and 15, after being included here in line 10. The rest of the program is written in exactly the same manner that it was written when we defined the structure locally. This program can be compiled and executed just like all of the other programs in this tutorial.

Programming Exercise:

---

1. Define a named structure containing a string for a name, an integer for feet, and another for arms. Use the new type to define an array of about 6 items. Fill the fields with data and print them out as follows.

```
A human being has 2 legs and 2 arms.
A dog has 4 legs and 0 arms.
A television set has 4 legs and 0 arms.
A chair has 4 legs and 2 arms.
```

2. Rewrite exercise 1 using a pointer to print the data out.
