# Sams Teach Yourself C in 24 Hours



# Hour 20 - Unions: Another Way to Collect Dissimilar Data

Coming together is a beginning;
keeping together is progress;
working together is success.

**—T. Roosevelt**

In the previous hour's lesson you learned how to store data of different types into structures. In this hour you'll learn another way to collect differently typed data items by using unions. You'll learn about the following topics in this lesson:

- How to declare and define unions
- How to initialize unions
- The differences between unions and structures
- Nested unions with structures
- Manipulating the bit field with struct

## What Is a Union?

A union is a block of memory that is used to hold data items of different types. In C, a union is similar to a structure, except that data items saved in the union are overlaid in order to share the same memory location. More details on the differences between unions and structures are discussed in the following sections.

### Declaring Unions

The syntax for declaring a union is similar to the syntax for a structure. The following is an example of a union declaration:

```
union automobile {
     int year;
     char model[8];
     int engine_power;
     float weight;
};
```

Here union is the keyword that specifies the union data type. automobile is the tag name of the union. The variables, such as year, model, engine_power, and weight, are members of the union, and are declared within the braces ({ and }). The union declaration ends with a semicolon (;).

Like a structure tag name, a union tag name is a label to a union, which is used by the compiler to identify the union.

### Defining Union Variables

You can define union variables after declaring a union. For instance, the following union variables are defined with the union labeled with automobile from the previous section:

```
union automobile sedan, pick_up, sport_utility;
```

Here the three variables, sedan, pick_up, and sport_utility, are defined as the union variables.

Of course, you can declare a union and define variables of the union in a single statement. For instance, you can rewrite the previous union declaration and definition like this:

```
union automobile {
     int year;
     char model[8];
     int engine_power;
     float weight;
} sedan, pick_up, sport_utility;
```

Here three union variables, sedan, pick_up, and sport_utility, are defined by the union of automobile in which there are four members of different data types. If you declare a union and define variables of the union in a single statement, and there is no more union variable definition made with the union, you can omit the tag name of the union. For instance, the tag name automobile can be omitted in the union definition like this:

```
union {
     int year;
     char model[8];
     int engine_power;
     float weight;
} sedan, pick_up, sport_utility;
```

### Referring a Union with . or ->

As well as being used to reference structure members, the dot operator (.) can be used in referencing union members. For example, the following statement assigns the value of 1997 to one of the members of the sedan union:

```
sedan.year = 1997;
```

Here the dot operator is used to separate the union name sedan and the member name year. In addition, if you define a pointer ptr like this:

```
union automobile *ptr;
```

then you can reference one of the union members in the following way:

```
ptr->year = 1997;
```

Here the arrow operator (->)is used to reference the union member year with the pointer ptr.

The program in Listing 20.1 gives another example of how to reference and assign values to the members of a union.

**TYPE**

**Listing 20.1. Referencing the members of a union.**

```
1:  /* 20L01.c Referencing a union */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main(void)
6:  {
7:     union menu {
8:        char name[23];
9:        double price;
10:    } dish;
11:
12:    printf("The content assigned to the union separately:\n");
13:    /* reference name */
14:    strcpy(dish.name, "Sweet and Sour Chicken");
15:    printf("Dish Name: %s\n", dish.name);
16:    /* reference price */
17:    dish.price = 9.95;
18:    printf("Dish Price: %5.2f\n", dish.price);
19:
20:    return 0;
21: }
```

After running the executable 20L01.exe of the program in Listing 20.1, I have the following output shown on my screen:

**OUTPUT**

```
C:\app>20L01
The content assigned to the union separately:
Dish Name: Sweet and Sour Chicken
Dish Price: 9.95
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 20.1 is to show you how to reference union members with the dot operator.

Inside the main() function, a union, called dish, is first defined with the union data type of menu in lines 7_10. There are two members, name and price, in the union.

Then the statement in line 14 assigns the string "Sweet and Sour Chicken" to the character array name that is one of the union members. Note that the dish.name expression is used as the first argument to the strcpy() function in line 14. When the compiler sees the expression, it knows that we want to reference the memory location of name that is a member of the dish union.

The strcpy() function is a C function that copies the contents of a string pointed to by the function's second argument to the memory storage pointed to by the function's first argu-ment. I included the header file string.h in the program before calling the strcpy() function. (See line 3.)

Line 15 prints out the contents copied to the name array by using the dish.name expression one more time.

The statement in line 17 assigns the value 9.95 to the double variable price, which is another member for the dish union. Note that the dish.price expression is used to reference the union member. Then line 18 displays the value assigned to price by calling the printf() function and passing the dish.price expression as an argument to the function.

According to the results shown in the output, the two members of the dish union, name and price, have been successfully referenced and assigned corresponding values.

## Unions Versus Structures

You might notice that in Listing 20.1, I assigned a value to one member of the dish union, and then immediately printed out the assigned value before I moved to the next union member. In other words, I didn't assign values to all the union members together before I printed out each assigned value from each member in the union.

I did this purposely because of the reason that is explained in the following section. So keep reading. (In exercise 1 at the end of this lesson, you'll see a different output when you rewrite the program in Listing 20.1 by exchanging the orders between the statements in lines 15 and 17.)

### Initializing a Union

As mentioned earlier in this lesson, data items in a union are overlaid at the same memory location. In other words, the memory location of a union is shared by different members of the union at different times. The size of a union is equal to the size of the largest data item in the list

of the union members, which is large enough to hold any members of the union, one at a time. Therefore, it does not make sense to initialize all members of a union together because the value of the latest initialized member overwrites the value of the preceding member. You initialize a member of a union only when you are ready to use it. The value contained by a union is the value that is latest assigned to a member of the union.

For instance, if we declare and define a union on a 16-bit machine (that is, the int data type is 2 bytes long) like this:

```
union u {
   char ch;
   int x;
} a_union;
```

then the following statement initializes the char variable ch with the character constant `H':

```
a_union.ch = `H';
```

and the value contained by the a_union union is the character constant `H'. However, if the int variable x is initialized by the following statement:

```
 a_union.x = 365;
```

then the value contained by the a_union union becomes the value of 365. Figure 20.1 demonstrates the content change of the union during the two initializations.

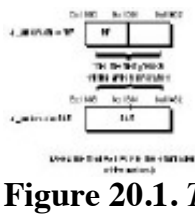

**Figure 20.1.** *The content of the a_union union is the same as the content assigned to one of its members.*

According to the ANSI C standard, a union can be initialized by assigning the first union member with a value. For instance, in the following statement:

```
union u {
   char ch;
   int x;
} a_union = {`H'};
```

the a_union union is said to be initialized because the character constant `H' is assigned to the first union member, ch.

If the first member of a union is a structure, the entire structure has to be initialized with a list of values before the union can be said to have been initialized.

Let's see what will happen if we try to assign values to all members of a union together. Listing 20.2 gives such an example.

**TYPE**

**Listing 20.2. The members of a union share the same memory location.**

```
1:  /* 20L02.c:  Memory sharing in unions */
2:  #include <stdio.h>
3:
4:  main(void)
5:  {
6:     union employee {
7:        int start_year;
8:        int dpt_code;
9:        int id_number;
10:    } info;
11:
12:    /* initialize start_year */
13:    info.start_year = 1997;
14:    /* initialize dpt_code */
15:    info.dpt_code = 8;

16:    /* initialize id */
17:    info.id_number = 1234;
18:
19:    /* display content of union */
20:    printf("Start Year:  %d\n", info.start_year);
21:    printf("Dpt. Code:   %d\n", info.dpt_code);
22:    printf("ID Number:   %d\n", info.id_number);
23:
24:    return 0;
25: }
```

After the executable 20L02.exe is created and executed, the following output is displayed on the screen:

**OUTPUT**

```
C:\app>20L02
Start Year:  1234
Dpt. Code:   1234
ID Number:   1234
C:\app>
```

As you can see in Listing 20.2, a union called info has three int variable members, start_year, dpt_code, and id_number. (See lines 6_10.) Then, these three union members are assigned with different values consecutively in lines 13, 15, and 17. And in lines 20_22, we try to print

out the values assigned to the three members. However, the output shows that every member in the info union has the same value, 1234, which is the integer assigned to the third member of the union, id_number. Note that id_number is the member that is assigned with 1234 last; the info union does hold the latest value assigned to its members.

### The Size of a Union

You've been told that the members of a union share the same memory location. The size of a union is the same as the size of the largest member in the union.

In contrast with a union, all members of a structure can be initialized together without any overwriting. This is because each member in a structure has its own memory storage. The size of a structure is equal to the sum of sizes of its members instead of the size of the largest member.

How do I know whether all these are true? Well, I can prove it by measuring the size of a union or a structure. Listing 20.3 contains a program that measures the size of a union as well as the size of a structure. The structure has exactly the same members as the union.

**TYPE**

**Listing 20.3. Measuring the size of a union.**

```
1:  /* 20L03.c The size of a union */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main(void)
6: {
7:    union u {
8:       double x;
9:       int y;
10:    } a_union;
11:
12:    struct s {
13:       double x;
14:       int y;
15:    } a_struct;
16:
17:    printf("The size of double: %d-byte\n",
18:            sizeof(double));
19:    printf("The size of int:    %d-byte\n",
20:            sizeof(int));
21:
22:    printf("The size of a_union:  %d-byte\n",
23:            sizeof(a_union));
24:    printf("The size of a_struct: %d-byte\n",
25:            sizeof(a_struct));
26:
27:    return 0;
28: }
```

The compiler on your machine may generate several warning messages, something like "unreferenced local variables." This is because the a_union union and the a_struct structure are not initialized in the program. You can ignore the warning messages. The following output is displayed on the screen after the executable 20L03.exe is created and executed:

**OUTPUT**

```
C:\app>20L03
The size of double: 8-byte
The size of int:    2-byte
The size of a_union:  8-byte
The size of a_struct: 10-byte
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 20.3 is to show the difference between a union memory allocation and a structure memory allocation, although both the union and the structure consist of the same members.

A union, called a_union, is defined in lines 7_10; it has two members, a double variable x and an int variable y. In addition, a structure, called a_structure and defined in lines 12_15, also consists of two members, a double variable x and an int variable y.

The statements in lines 17_20 first measure the sizes of the double and int data types on the host machine. For instance, on my machine, the size of the double data type is 8 bytes long and the int data type is 2 bytes long.

Then lines 22_25 measure the sizes of the a_union union and the a_structure structure, respectively. From the output, we see that the size of a_union is 8 bytes long. In other words, the size of the union is the same as the size of the largest member, x, in the union.

The size of the structure, on the other hand, is the sum of the sizes of two members, x and y, in the structure (10 bytes in total).

## Using Unions

Now let's focus on the applications of unions. Basically, there are two kinds of union applications, which are introduced in the following two sections.

### Referencing the Same Memory Location Differently

The first application of unions is to reference the same memory location with different union members.

To get a better idea about referencing the same memory with different union members, let's have a look at the program in Listing 20.4, which uses the two members of a union to reference the same memory location. (We assume that the char data type is 1 byte long, and the int data type is 2 bytes long, which are true on many machines.)

**TYPE**

**Listing 20.4. Referencing the same memory location with different union members.**

```
1:  /* 20L04.c: Referencing the same memory in different ways */
2:  #include <stdio.h>
3:
4:  union u{
5:     char ch[2];
6:     int num;
7:  };
8:
9:  int UnionInitialize(union u val);
10:
11: main(void)
12: {
13:    union u val;
14:    int x;
15:
16:    x = UnionInitialize(val);
17:
18:    printf("The two character constants held by the union:\n");
19:    printf("%c\n", x & 0x00FF);
20:    printf("%c\n", x >> 8);
21:
22:    return 0;
23: }
24: /* function definition */
25: int UnionInitialize(union u val)
26: {
27:    val.ch[0] = `H';
28:    val.ch[1] = `i';
29:
30:    return val.num;
31: }
```

The following output is printed on the screen after the executable 20L04.exe is created and executed:

**OUTPUT**

```
C:\app>20L04
The two character constants held by the union:
H
i
C:\app>
```

**ANALYSIS**

As you see from the program in Listing 20.4, a union called val is defined in line 13. It contains two members; one is a char array ch and the other is an int variable num. If a char data type is 1 byte long and an int data type is 2 bytes long, the ch array and the integer variable num have the same length of memory storage on those machines.

A function named UnionInitialize()is called and passed with the union name val in line 16. The definition of the UnionInitialize() function is shown in lines 25_31.

From the function definition, you can see that the two elements of the char array ch are initialized with two character constants, `H' and `i' (in lines 27 and 28). Because the char array ch and the int variable num share the same memory location, we can return the value of num that contains the same content as the ch array. (See line 30.) Here we've used the two members, ch and num, in the val union to reference the same memory location and the same contents of the union.

The value returned by the UnionInitialize() function is assigned to an int variable x in line 16 inside the main() function. The statements in lines 19 and 20 print out the 2 bytes of the int variable num. Each byte of num corresponds to a character that is used to initialize the ch array because num and ch are all in the same union and have the same content of the union. Line 19 displays the low byte of num returned by the x & 0x00FF expression. In line 20, the high byte of num is obtained by shifting the x variable to the right by 8 bits. That is, by using the shift-right operator in the x >> 8 expression. (The bitwise operator (&) and the shift operator (>>) are introduced in Hour 8, "More Operators.")

From the output, you can see that the content of the val union is shown on the screen correctly.

Figure 20.2 shows the locations of the two character constants in memory.

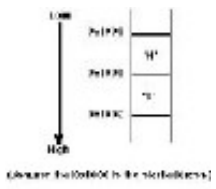

**Figure 20.2.** *The memory locations of the two character constants.*

**NOTE**

There are two formats to store a multiple-byte quantity, such as the int variable num in Listing 20.4. One of the formats is called the little-endian format; the other is the big-endian format.

For the little-endian format, the high bytes of a multiple-byte quantity are stored at higher memory addresses and the low bytes are saved at lower addresses. The little-endian format is used by Intel's 80x86 microprocessors. My computer's CPU is a

Pentium microprocessor, which is one of the members in the 80x86 family. Therefore, in Listing 20.4, the character constant `H', which is a low byte, is stored at the lower address. `i' is stored at the higher address because it's a high byte.

The big-endian format is just opposite. That is, the high bytes are stored at lower addresses; the low bytes are stored at higher addresses. Motorola's 68000 microprocessor family uses the big-endian format.

### Making Structures Flexible

The second application of unions is to nest a union inside a structure so that the structure can hold different types of values.

For example, suppose we want to write a program that asks the user about the name of a cable company or a satellite dish company that provides service to the user. Assume that the user either uses a cable or a satellite dish at home, but not both. Then, if we define two character arrays to store the cable company and satellite dish company names respectively, one of the arrays will be empty due to the assumption. In this case, we can declare a union with the two character arrays as its members so that the union can hold either a cable company name or a satellite dish company name, depending on the user's input. Listing 20.5 demonstrates how to write a program with such a union.

**TYPE**

**Listing 20.5. Making a structure flexible.**

```
1:  /* 20L05.c: Using unions */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  struct survey {
6:     char name[20];
7:     char c_d_p;
8:     int age;
9:     int hour_per_week;
10:    union {
11:        char cable_company[16];
12:        char dish_company[16];
13:    } provider;
14: };
15:
16: void DataEnter(struct survey *s);
17: void DataDisplay(struct survey *s);
18:
19: main(void)
20: {
21:    struct survey tv;
22:
23:    DataEnter(&tv);
24:    DataDisplay(&tv);
25:
26:    return 0;
27: }
28: /* function definition */
29: void DataEnter(struct survey *ptr)
30: {
31:    char is_yes[4];
32:
33:    printf("Are you using cable at home? (Yes or No)\n");
34:       gets(is_yes);
35:    if ((is_yes[0] == `Y') ||
36:        (is_yes[0] == `y')){
37:       printf("Enter the cable company name:\n");
38:       gets(ptr->provider.cable_company);
39:       ptr->c_d_p = `c';
40:    } else {
41:       printf("Are you using a satellite dish? (Yes or No)\n");
42:          gets(is_yes);
43:       if ((is_yes[0] == `Y') ||
44:           (is_yes[0] == `y')){
45:          printf("Enter the satellite dish company name:\n");
46:          gets(ptr->provider.dish_company);
47:          ptr->c_d_p = `d';
48:       } else {
49:          ptr->c_d_p = `p';
50:       }
51:    }
52:    printf("Please enter your name:\n");
53:       gets(ptr->name);
54:    printf("Your age:\n");
55:       scanf("%d", &ptr->age);
56:    printf("How many hours you spend on watching TV per week:\n");
57:       scanf("%d", &ptr->hour_per_week);
58: }
59: /* function definition */
60: void DataDisplay(struct survey *ptr)
61: {
62:    printf("\nHere's what you've entered:\n");
63:    printf("Name: %s\n", ptr->name);
64:    printf("Age:  %d\n", ptr->age);
65:    printf("Hour per week: %d\n", ptr->hour_per_week);
66:    if (ptr->c_d_p == `c')
67:       printf("Your cable company is: %s\n",
68:          ptr->provider.cable_company);
69:    else if (ptr->c_d_p == `d')
```

```
70:       printf("Your satellite dish company is: %s\n",
71:          ptr->provider.dish_company);
72:    else
73:       printf("You don't have cable or a satellite dish.\n");
74:    printf("\nThanks and Bye!\n");
75: }
```

When the executable program 20L05.exe is being run, I enter my answers to the survey and have the following output displayed on the screen (my answers are shown in bold monospace type in the output):

**OUTPUT**

```
C:\app>20L05
Are you using cable at home? (Yes or No)
No
Are you using a satellite dish? (Yes or No)
Yes
Enter the satellite dish company name:
ABCD company
Please enter your name:
Tony Zhang
Your age:
30
How many hours you spend on watching TV per week:
8

Here's what you've entered:
Name: Tony Zhang
Age: 30
Hour per week: 8
Your satellite dish company is: ABCD company

Thanks and Bye!
C:\app>
```

**ANALYSIS**

As you can see in lines 5_14, a structure data type with the tag name survey is declared, and in it a nested union called provider has two members, the cable_company array and the dish_company array. The two members of the union are used to hold the names of cable or satellite dish companies, depending on the user's input.

The statements in lines 16 and 17 declare two functions, DataEnter() and DataDisplay(), in which a pointer with struct survey is passed to the two functions as the argument.

A structure called tv is defined in line 21 inside the main() function. Then in lines 23 and 24, the DataEnter() and DataDisplay() functions are invoked with the address of the tv structure as their argument.

Lines 29_58 contain the definition of the DataEnter() function, which asks the user to enter proper information based on the survey questions. Under the assumption we made earlier, the user can use either cable or a satellite dish, but not both. If the user does use cable, then line 38 receives the cable company name entered by the user and saves it into the memory storage referenced by one of the members in the provider union, cable_company.

If the user uses a satellite dish, then line 46 stores the satellite dish company name entered by the user into the same location of the provider union. But this time the name of another union member, dish_company, is used to reference the memory location. Now you see how to save the memory by putting two exclusive data items into a union.

In fact, the program supports another situation in which the user has neither cable nor a satellite dish. In this case, the char variable c_d_p, which is a member of the structure, is assigned with the character constant `p'.

Lines 60_75 give the definition of the DataDisplay() function that prints out the information entered by the user back to the screen. The output shown here is a sample I made by running the executable program of Listing 20.5 on my machine.

**Defining Bit Fields with struct**

In this section, we'll revisit our old friend the struct keyword to declare a very small object. Then we'll use the object with unions.

As you know, char is the smallest data type in C. On many machines, the char data type is 8 bits long. However, with the help of the struct keyword, we can declare a smaller object—a bit field—which allows you to access a single bit. A bit is able to hold one of the two values, 1 or 0.

The general form to declare and define bit fields is

```
struct tag_name {
   data_type name1: length1;
   data_type name2: lenght2;
   . . .
   data_type nameN: lengthN;
} variable_list;
```

Here the struct keyword is used to start the declaration. tag_name is the tag name of the struct data type. data_type, which must be either int, unsigned, or signed, specifies the data type of bit fields. names1, name2, and nameN are names of bit fields. length1, length2, and lengthN indicate the lengths of bit fields, which may not exceed the length of the int data type. variable_list contains the variable names of the bit field.

For instance, the following statement defines a variable called jumpers with three bit fields:

```
struct bf {
   int jumper1: 1;
   int jumper2: 2;
   int jumper3: 3;
} jumpers;
```

Here jumper1, jumper2, and jumper3 are the three bit fields with lengths of 1 bit, 2 bits, and 3 bits, respectively. Figure 20.3 demonstrates the memory allocations of the three bit fields.

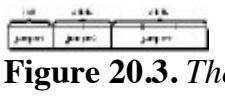

**Figure 20.3.** *The memory allocations of jumper1, jumper2, and jumper3.*

The program in Listing 20.6 is an example of using the bit fields defined with struct. In fact, the program in Listing 20.6 is a modified version of the program in Listing 20.5.

**TYPE**

**Listing 20.6. Applying bit fields.**

```
1:  /* 20L06.c: Applying bit fields */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  struct bit_field {
6:     int cable: 1;
7:     int dish: 1;
8:  };
9:
10: struct survey {
11:    char name[20];
12:    struct bit_field c_d;
13:    int age;
14:    int hour_per_week;
15:    union {
16:        char cable_company[16];
17:        char dish_company[16];
18:    } provider;
19: };
20:
21: void DataEnter(struct survey *s);
22: void DataDisplay(struct survey *s);
23:
24: main(void)
25: {
26:    struct survey tv;
27:
28:    DataEnter(&tv);
29:    DataDisplay(&tv);
30:
31:    return 0;
32: }
33: /* function definition */
34: void DataEnter(struct survey *ptr)
35: {
36:    char is_yes[4];
37:
38:    printf("Are you using cable at home? (Yes or No)\n");
39:       gets(is_yes);
40:    if ((is_yes[0] == `Y') ||
41:        (is_yes[0] == `y')){
42:       printf("Enter the cable company name:\n");
43:       gets(ptr->provider.cable_company);
44:       ptr->c_d.cable = 1;
45:       ptr->c_d.dish = 0;
46:    } else {
47:       printf("Are you using a satellite dish? (Yes or No)\n");
48:          gets(is_yes);
49:       if ((is_yes[0] == `Y') ||
50:           (is_yes[0] == `y')){
51:          printf("Enter the satellite dish company name:\n");
52:          gets(ptr->provider.dish_company);
53:          ptr->c_d.cable = 0;
54:          ptr->c_d.dish = 1;
55:       } else {
56:          ptr->c_d.cable = 0;
57:          ptr->c_d.dish = 0;
58:       }
59:    }
60:    printf("Please enter your name:\n");
61:       gets(ptr->name);
62:    printf("Your age:\n");
63:       scanf("%d", &ptr->age);
64:    printf("How many hours you spend on watching TV per week:\n");
65:       scanf("%d", &ptr->hour_per_week);
66: }
67: /* function definition */
68: void DataDisplay(struct survey *ptr)
69: {
70:    printf("\nHere's what you've entered:\n");
71:    printf("Name: %s\n", ptr->name);
72:    printf("Age:  %d\n", ptr->age);
73:    printf("Hour per week: %d\n", ptr->hour_per_week);
74:    if (ptr->c_d.cable && !ptr->c_d.dish)
```

```
75:       printf("Your cable company is: %s\n",
76:          ptr->provider.cable_company);
77:    else if (!ptr->c_d.cable && ptr->c_d.dish)
78:       printf("Your satellite dish company is: %s\n",
79:          ptr->provider.dish_company);
80:    else
81:       printf("You don't have cable or a satellite dish.\n");
82:    printf("\nThanks and Bye!\n");
83: }
```

Because the program in Listing 20.6 is basically the same as the one in Listing 20.5, I have the same output shown on the screen after I run the executable 20L06.exe and enter the same answers to the survey:

**OUTPUT**

```
C:\app>20L06
Are you using cable at home? (Yes or No)
No
Are you using a satellite dish? (Yes or No)
Yes
Enter the satellite dish company name:
ABCD company
Please enter your name:
Tony Zhang
Your age:
30
How many hours you spend on watching TV per week:
8

Here's what you've entered:
Name: Tony Zhang
Age: 30
Hour per week: 8
Your satellite dish company is: ABCD company

Thanks and Bye!
C:\app>
```

**ANALYSIS**

The purpose of the program in Listing 20.6 is to show you how to declare bit fields and how to use them. As you can see in lines 5_8, two bit fields, cable and dish, are declared with the struct data type. Each of the bit fields is 1 bit long. Then a structure called c_d is defined with the two bit fields in line 12, which is within another structure declaration from line 10 to line 19.

The bit fields cable and dish are used as flags to indicate whether the user is using cable or a satellite dish based on the answers made by the user. If the user has cable, then the cable bit field is set to 1 and the dish bit field is set to 0. (See lines 44 and 45.) On the other hand, if the user has a satellite dish, then dish is set to 1 and cable is set to 0, as shown in lines 53 and 54. If, however, the user has neither cable nor a satellite dish, both cable and dish are set to 0 in lines 56 and 57.

So you see, we've used the combinations of the two bit fields, cable and dish, to represent the three situations: having cable, having a satellite dish, or having neither cable nor a satellite dish.

Since the program in Listing 20.6 is basically the same as the one in Listing 20.5, I get the same output after I run the executable program of Listing 20.6 and enter the same information as I did to the executable 20L05.exe.

## Summary

In this hour you've learned the following:

- A union is a block of memory that is used to hold data items of different types.
- A union is similar to a structure, except that data items saved in the union are overlaid in order to share the same memory location.
- The size of a union is the same as the size of the largest member in the union.
- The union keyword has to be used to specify the union data type in a union declaration or a union variable definition.
- To reference a union member, you can use either a dot operator (.) to separate the union name and the union member name or an arrow operator (->) to separate the name of a pointer that points to the union and the union member name.
- The ANSI C standard allows you to initialize a union by assigning the first union member a value.
- You can access the same memory location with different union members.
- To make a structure flexible, you can nest a union inside a structure so that the structure can hold different types of values.
- You can define the bit fields, which can be a single bit or any number of bits up to the number of bits in an integer, by using the struct data type.

In the next hour you'll learn how to read from or write to disk files.

## Q&A

**Q** What are the differences between a union and a structure?

**A** Basically, the difference between a union and a structure is that the members in a union are overlaid and they share the same memory location, whereas the members in a structure have their own memory locations. A union can be referenced by using one of its member names.

**Q** What will happen if you initialize all members of a union together?

**A** The value that is assigned to a union member last will be the value that stays in the memory storage of the union until the next assignment to the union. In ANSI C, you can initialize a union by initializing its first member.

**Q** How do you reference a union member?

**A** If the name of a union is used to reference the union members, then the dot operator (.) can be used to separate the union name and the name of a union member. If a pointer, which points to a union, is used to reference the union members, then the arrow operator (->) can be used between the pointer name and the name of a union member.

structure.

Can you access the same memory location with different union members?

**A** Yes. Since all union members in a union share the same memory location, you can access the memory location with different union members. For example, in the program in Listing 20.4, two character constants are assigned to a union memory storage through one of the union members. The two characters saved at the memory location of the union are printed out with the help from another union member.

## Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

### Quiz

1. Of the following two statements, which one is the declaration of a union and which one is the definition of union variables?

```
union a_union {
     int x;
     char y;
     };
```

```
union a_union x, y;
```

2. What's wrong with the following union declaration?

```
union automobile {
     int year;
     char model[8]
     float weight;
     }
```

3. In the following statement, what are the values contained by the two union members?

```
union u {
          int date;
     char year;
} a_union = {1997};
```

4. Given a structure and a union, what are the sizes of the structure and the union? (Assume that the char data type is 1 byte long and the int data type is 2 bytes long.)

```
struct s {
     int x;
     char ch[4];
} a_structure;

union u {
     int x;
     char ch[4];
} a_union;
```

### Exercises

1. Rewrite the program in Listing 20.1 by switching the order between the statement in line 15 and the statement in line 17. What do you get after running the rewritten program?
2. Rewrite the program in Listing 20.2. This time, print out values held by all the members in the info union each time one of the members is assigned an integer.
3. Write a program to ask the user to enter his or her name. Then ask the user whether he or she is a U.S. citizen. If the answer is Yes, ask the user to enter the name of the state where he or she comes from. Otherwise, ask the user to enter the name of the country he or she comes from. (You're required to use a union in your program.)
4. Modify the program you wrote in exercise 3. Add a bit field and use it as a flag. If the user is a U.S. citizen, set the bit field to 1. Otherwise, set the bit field to 0. Print out the user's name and the name of country or state by checking the value of the bit field.