Lesson 11

STRUCTURES

This lesson introduces the basics for the type structure. The C language allows you to create new data types by combining many variables into one conglomerate variable called a structure. You can also create new names for standard variable types using typedef. A structure is a collection of variables that is referenced using a user-defined name. This allows you to define and use a structure to provide a convenient means of keeping related information in one place.

A structure definition forms a template that may be used to create structure variables. Each structure is made up of one or more variables that are logically related. These variables are called structure elements. Structures can be passed easily to functions. For example, a name and addressess grouping in a mailing list is a common set of related information. The following code fragment declares a structure to hold the name and address fields; the keyword struct tells the compiler that a structure template is being defined:

```
struct address {
        char name[30];
        char street[40];
        char city[20];
        char state[2];
        unsigned long int zip;
};
```

There are two important aspects about the definition. First, a semicolon terminates the structure. This is because a structure definition is a C statement. Second, the structure tag "address" identifies the user defined data structure. The tag is the name that you will uses in the program to declare a new variable.

At this point you not actually declared an actual variable. You have only defined the form of the data structure. To declare an actual variable with this structure, you would write

```
struct address address_info;
```

This will declare a variable of type address called address_info. When you define a structure, you are in really defining a complex variable type made up of structure elements. You may also declare one or more variables at the same time that you define as a structure. For example:

```
struct address {
        char name[30];
        char street[40];
        char city[20];
        char state[2];
        unsigned long int zip;
    } address_one, address_two, address_three;
```

This will define a structure called address and declare three variables address_one, address_two, and address_three of type address.
If you only need one structure variable, the structure name is not needed.

That is an example:

```
    struct  {
        char name[30];
        char street[40];
        char city[20];
        char state[2];
        unsigned long int zip;
    } address_info;
```

Notice that the structure name is missing and struct declares only one, variable called address_info of the structure preceding it.

The general form of a structure definition is:

```
    struc structure__name {
        type variable__name;
        type variable__name;
        type variable  name;

    } structure__varables;
```

If you notice in the general form either the structure__name or the structure__variables may be omitted, but not both.

Referencing Structure Elements

The following code will assign the ZIP code 12345 called zip to the structure variable address_info declared earlier.

```
address_info.zip = 12345;                    (This the member element )
```

As you can see, the structure variable name address_info followed by a period and the element name will reference that individual structure element. C programmers often call the period the dot operator, but it essentially signifies that a structure element follows. All structure elements are accessed in the same way. The general form is

```
structure__name.element__name
```

Therefore, to print the ZIP code to the screen, you could write

```
printf("%d",address_info.zip);
```

This will print the ZIP code contained in the variable zip of the structure variable address_info.

For example, consider address_info.name**.** This element is an array of characters. Using gets() to input a name, you would write

```
gets(address_info.name);
```

This will pass a character pointer to the start of name. If you wished to access the individual elements of address_info.name, you could index name. For example, you could print the contents of address_info.name by using

```
        int t;
        for (t=0; address_info.name[t]; ++t)
                putchar(address_info.name[t]);
```

Arrays of Structures

Perhaps the most common usage of structures is an array of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type address you would write:

```
struct address address_info[100];
```

This creates 100 sets of variables that are organized as defined in the structure address. To print the ZIP code of structure 3, you would write

```
printf("%d",address_info[2].zip);
```

Remember like all array variables, an array of structures also begin their indexing at zero.

A Mailing List Example

In this example, a simple Mailing List Program will be developed that uses an array of structures to hold the street address information. The functions in this program interact with structures and their elements to illustrate structure usage. In this example, the information that will be stored includes

```
        .name
        .street
```

.state
.zip code

To define the basic data structure, address**,** that will hold this information, you would write

```
struct address {
char name[30];
char street[40];
char city[20];
char state[3];
unsigned long int zip;
};
```

Notice that the ZIP code field is an unsigned long integer. This is done because ZIP codes greater than 64000 - such as
94564 - cannot be represented in a two-byte integer. In this example, an integer is used to hold the ZIP code to illustrate a numeric structure element; however, the more common practice is to use a character string that accommodates ZIP codes with letters, as will as numbers. Once the data structure has been defined, you can declare an array of structures with the following statement:

```
struct address address_info[100];
```

This declares an array called **address_info,** which contains 100 structures to type **address.** The first function needed for the program is **main():**

```
main() /* simple mailing list example suing structures */
{       int menu_select();
void init_list(), delete(), list(),enter();
char s[80], choice;

init_list(); /* initialize the structure array*/
do{
choice=menu_select();
switch(choice) {
case 1: enter();
break;
case 2: delete();
break;
case 3: list();
break;
case 4: exit(O);
}
} whiled);
}
```

Here the function **init__list ()** prepares the structure array for use by putting a null character into the first byte of the name field. The program assumes that if the name field is empty, that structure variable is not in use. The **init__list ()** function is written as

```
void init__list()
{
register int t;
for(t=0;t<100;++t)address_info[t].name[0]='\0';
```

Next the **menu__select()** function will display the option messages and return the user's selection:

```
int menu__select()
{
char s[80];
int c=-99:

printf("1. Enter a name\n");
printf("2. Delete a name\n");
printf("3. List the file\n");
printf("4. Quit\n");
do{
Printf("\nEnter your choice: ");
gets(s);
c=atoi(s);
}while(c<1 U c>4)
return c:
```

The enter( ) function prompts the user for input and places the information entered into the next free structure. If the array is full, the message list full is printed on the screen. The find__free( ) function searches the structure array for an unused element. Both functions are written as

```
void enterO
{
int slot;
slot=find__free();
if(slot==-1) {
printf("\nlist full");
return;
}


Printf("enter name: ");
gets(address_info[slot].name);
printf("enter street: ");
gets(address_info[slot].street);
printf("enter city: ");
gets(address_info[slot].city);
printf("enter state: ");
gets(address_info[slot].state);
printf("enter zip: ");
scanf(" %ld",&address_info[slot].zip);
}
find__free()
{
register int t;
for(t=0;address_info[t].name[0]=='\0' &&t<100;++t);
```

```
if(t==100) return -1; /* to return the first aviable free one */
return t:
```

Notice that find__free() returns a -1 if every structure array variable is in use. This is a safe number to use because they cannot be a -1 element.

The deleted function simply requires the user to specify the number of the street address that needs to be deleted.

The function then puts a null character in the first character position of the name field.

```
void delete( )
        {
        register int slot;
        char s[80];
        printf("enter record #: ")',
        gets(s);
        slot=atoi(s)
        if(slot>0 &&slot<100) address_info[slot].name[0]='\0';
```

The final function the program needs is list(), which prints the entire mailing list on the screen

The Source for
Client / Server Training

```
void list()
{
register int t;
for(t=0;t<100;++t) {
if(address_info[t].name[0]) {

printf("/os\n",address_info[t].name);
printf("%s\n",address_info[t].street);
printf("%s\n",address_info[t].city);
printf("%s\n",address_info[t].state);
printf("%u\n",address_info[t].zip);

}
printf("\n\n");
```

Passing Structure Elements to Functions

When you pass an element of a structure variable to a function, you are actually passing the value of that element to the function. Therefore, you are passing a simple variable. For example,

```
struct fred {
char x;
inty;
float z;
char s[10];
} mike;
```

Here are examples of each element being passed to a function;

```
func(mike.x); /* passes character value of x */
func2(mike.y); /* passes integer value of y */
func3(mike.z); /* passes float value of z */
func4(mike.s); /* passes address value of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

However, if you wished to pass the address of individual structure elements, you would place the & operator before the structure name. For example, to pass the address of the elements in the structure mike, you would write

```
func(&mike.x); /* passes the address of the character value of x */
func2(&mike.y); /* passes the address of the integer value of y */
func3(&mike.z); /* passes the address of the float value of z */
func4(mike.s); /* passes the address value of string (s), because it is and address */
func(&mike.s[2]); /* passes the address of the character value of s[2] */
```

Notice that the & operator precedes the structure name, not the individual element name.

Passing Entire Structures to Functions

When a Structure is passed to a function, only the address of the first byte of the structure is passed. This is similar to the way arrays are passed to functions. It is not feasible to copy the entire structure each time it is passed to a function; therefore, only its address is transferred. Because the function will be referencing the actual structure and not a copy, you will be able to modify the contents of the actual elements of the structure used in the call.

The general concept behind passing structure to a function is that an address is passed. This means that you will be working with a pointer to a structure similar to the way you work with a pointer to an array. For example,

```
if((*t).hours==24) (*t).hours=0;
```

This line of code tells the compiler to take the address oft and assign zero to its element called hours. The parentheses are necessary around the *t because the dot operator has a higher priority than the *. In actual practice, however , you will seldom, if ever, see references to a structure passed to a function as in the example just given. The reason is that this type of structure-element accessing is so common that a special operator is defined by C to perform this task. It is the **->.** Most C programmers call this the arrow operator. It is formed by using the minus sign allowed by a greater-than sign. The **->** is used in place of the dot operatorwhen accessing a structure element inside a function. For example,

```
(*t).hours
```

is the same as

```
t->hours
```

You use the dot operator to access structure elements when the structure is either global or defined inside the same function as the code referencing it. You use the -> to reference structure elements when a structure pointer has been passed to a function.

Also remember that you have to pass the address of the structure to a function using the & operator, unless a array struct is used. Structures are not like arrays, which can be passed by the array name only, unless a array struct is used.