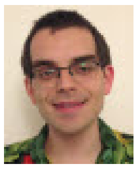
Lesson 7: Structures in C



By Alex Allain

When programming, it is often convenient to have a single name with which to refer to a group of a related values. Structures provide a way of storing many different values in variables of potentially different types under the same name. This makes it a more modular program, which is easier to modify because its design makes things more compact. Structs are generally useful whenever a lot of data needs to be grouped together--for instance, they can be used to hold records from a database or to store information about contacts in an address book. In the contacts example, a struct could be used that would hold all of the information about a single contact--name, address, phone number, and so forth.

The format for defining a structure is

```
struct Tag {
  Members
};
```

Where Tag is the name of the entire type of structure and Members are the variables within the struct. To actually create a single structure the syntax is

```
struct Tag name_of_single_structure;
```

To access a variable of the structure it goes

```
name_of_single_structure.name_of_variable;
```

For example:

```
struct example {
  int x;
};
struct example an_example; /* Treating it like a normal variable type
                            except with the addition of struct*/
an_example.x = 33;          /*How to access its members */
```

Here is an example program:

```
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  struct database employee;  /* There is now an employee variable that has
                              modifiable variables inside it.*/
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

The struct database declares that it has three variables in it, age, id_number, and salary. You can use database like a variable type like int. You can create an employee with the database type as I did above. Then, to modify it you call everything with the 'employee.' in front of it. You can also return structures from functions by defining their return type as a structure type. For instance:

```
struct database fn();
```

I will talk only a little bit about unions as well. Unions are like structures except that all the variables share the same memory. When a union is declared the compiler allocates enough memory for the largest data-type in the union. It's like a giant storage chest where you can store one large item, or a small item, but never the both at the same time.

The '.' operator is used to access different variables inside a union also.

As a final note, if you wish to have a pointer to a structure, to actually access the information stored inside the structure that is pointed to, you use the -> operator in place of the . operator. All points about pointers still apply.

A quick example:

```
#include <stdio.h>

struct xampl {
  int x;
};

int main()
{
    struct xampl structure;
    struct xampl *ptr;

    structure.x = 12;
    ptr = &structure; /* Yes, you need the & when dealing with
                           structures and using pointers to them*/
    printf( "%d\n", ptr->x );  /* The -> acts somewhat like the * when
                                   does when it is used with pointers
                                    It says, get whatever is at that memory
                                   address Not "get what that memory address
                                   is"*/
    getchar();
}
```

Still not getting it? Ask an expert!
Quiz yourself
Previous: Pointers
Next: Arrays
Back to C Tutorial Index