**STRUCTURES**

Structures are derived data types—they’re constructed using objects of other types. Structure is also a variable. A Structure can also be viewed as a container to hold many types of variables. Unlike arrays which hold only one type of variable, structures enable us to store multiple variables.

The various topics we will talk are

* Declare them with and without Tags
* Access members of structures
* Initialize structures
* Computing Size of structures, use sizeof operator
* Assign one structure to another
* Compare one structure with another - X, compare
* Pass by Value (default)
* Pass by Reference - send the address
* Having Pointers as members
* accessing structures using pointers
* accessing array of structures

we define a structure type as,

struct \_point {

int x ; // member variables

int y ; // member variables

} ;

Here struct is a keyword to describe a structure variable. x and y are member variables , each is a type of int. \_point is a tag. Structure definitions should always end semicolon.

Member variables need not be same type too. Consider this structure,

struct \_profile {

int age;

char name [ 10 ] ;

} ;

In the above definition, we have a char variable and an int variable as members. \_profile is a tag.

NOTE: There is no memory allocated to the structure, because it is just a type definition.

To define new structure variables using the above definitions, we could do

struct \_point point;

struct \_profile user1;

Here, point is a structure variable and user1 is also a structure variable.

So, what is the type of the variable - point ? It is struct \_point and the type of user1 is struct \_profile .

Using the definition of the struct \_point , I can define new variables as

struct \_point pt1, pt2, \*ptr ;

In the above definition, we have pt1 and pt2 as structure variables and ptr is a pointer of type struct \_point .

NOTE: No two variables of a struct can have the same name.

**How do we access member variables using DOT operator**

The Dot operator is represented by the symbol . To write values to the member variables, we use the DOT operator, like this

pt1.age = 20 ;

strcpy ( pt1.name, “John”) ; // Note we cannot do pt1.name = "John"

similarly , we can read values of the variables like

int x = pt1.age ;

char myAge [10 ] ;

strcpy ( myAge, pt1.name) ;

Although structure names should be distinct, we may use the same name for members in different structures. By using the DOT operator, we can access the member variables uniquely.

The dot operator is mandatory to access a member variable even if the name of the variable is not defined elsewhere.

Declaring structures with and without tag names.

Consider

struct {

int x ;

int y ;

} x, y, z ;

is same as

struct \_point {

int x ;

int y ;

} x, y, z ;

But the latter method with tag name is better and preferred as we can declare more variables like this

struct \_point a, b, c ;

The other advantage is you can pass the struct variable to functions as the type is struct \_point ;

The disadvantage with declaring structures without tags is

* you cannot define new variables and
* you cannot pass them to functions.

***IT IS ALWAYS RECOMMENDED TO DECLARE STRUCTURES WITH TAG NAMES***

**Initialize Structure**

We will discuss several methods to assign individual members with values.

Method 1: This is tedious way of initializing.

// method 1 : initialize individual members

pt1.age = 40;

strcpy ( pt1.name, "John") ;

In the above method, we use the dot operator. Pay attention to the use of strcpy method to copy.

Method 2: We could also initialize the members using declaration such as

struct \_profile p1 = { 30, "John" } ; // order is important

As discussed, this method depends on the order of definition of the member variables, compiler will not make a guess if you mix the order like

struct \_profile p1 = { "John" , 30} ; // this is incorrect.

That will result in compiler warning

Method 3: The other method is to assign one structure to another structure

struct \_profile p4 = p1 ; // is already initialized, so we copy into p4

technically, this copies bit by bit from p1 to p4.

The last method is of course , we use the scanf function to initialize the initialize each and every individual members. but this is not recommended.

**Sizeof Structures**

One possible gotcha with structures is , you cannot sum the size of individual members to determine the size of a structure. It is machine dependent, it creates members on the word boundary, especially if it contains non-char type members. Generally , it is 4 bytes.

Let us consider a simple case:

struct \_user {

int age ;

char name[12] ;

} ;

The size of this structure can readily be determined to be : 20 bytes , because int ( 4 bytes) plus 12 chars. This is very trivial.

But not so when it contains char and int variables , such as

struct \_user {

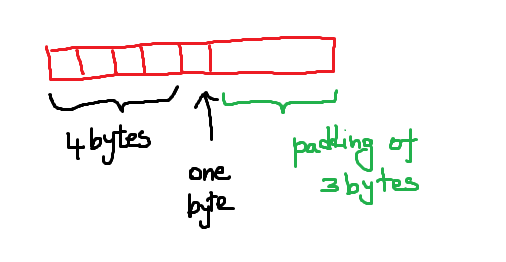
int age ;

char gender;

} ;

In this case, it is not 5 bytes, it could be 8 bytes (or 6 bytes in some sysems) with a padding of 3 additional bytes next to the gender variable.

Here is the picture :



But if you change the order of the members

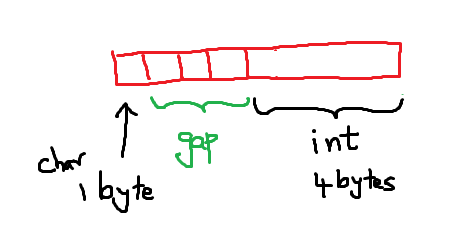
struct \_person {

char gender ;

int age ;

} ;

there will be a gap of 3 bytes.



The values in these gaps and paddings will be garbage. This is one of the main reasons you cannot compare two similar structures.

**ASSIGN STRUCTURES and COMPARE STRUCTURE**

Because structures have gaps or paddings, and these have garbage values, we could assign a structure variable to another similar type structure as it just a bit by bit copy. But you cannot compare two similar structures because during initialization it may have garbage values. But we cannot compare structures using any of the relation operators such as ==, <=, <, >, >=

**Using Typedef structures**

C provides the typefef construct , which lets the programmer provide a synonym for either a built-in or user defined data type. Although typedef may be used with any data tye, structures are generally used with typedef. typedef are just an alias to a type. For instance,

typedef int MyINT ;

MyInt becomes a synonym for int. Subsequently, we can declare new int type variables as

MyINT age ;

In the above definition age is a variable of type MyINT. Note, the syntax of a typedef: First comes the keyword typedef, then the data type and followed by the user provided name for this data type.

***A typedef is used only to create a synonym for a data type. By defining a tyepdef, we are not allocating any memory.***

Similarly, we could define a structure using typedef as

typedef struct {

int x, y ;

} Point\_t ;

To define a new variable - point, we simply say

Point\_t p1, \*ptr ;

It is not generally recommended to define a structure without a tag name though typedef definition above is sufficient. So, let us refine the above definition as

typedef struct \_point {

int x, y ;

} Point\_t ;

Now, I can define variables as

Point\_t p1, p2, \*ptr ;

The above definition is generally used in nested structures as we will see soon.

***Nested Structure***

Structures can have other type of structures as members , this is known as nested structures.

typedef struct \_cars {

char make [ 12 ];

char model [ 12 ];

} Cars\_t;

typedef struct \_person {

char name[12];

int age;

Cars\_t cars [ 2] ;

char ch; // padding of 3 bytes , each cell

} Person\_t;

Pointers to Structures

C provides pointers to structure variables and a special pointer operator **->** for accessing members of structures.

Consider this structure

typedef struct \_ram {

float price;

int size ;

} Ram\_t ;

Ram\_t p1, p2, \*ptr;

we have two structure variables p1 and p2 of type Ram\_t . We also have a pointer ptr defined.

/\* we assign the address of p1 to ptr \*/

ptr = &p1 ;

/\* assign values to price and ramSize \*/

(\*ptr).price = 43.99;

( \*ptr).size = 32 ; // GB

Some explanation : First we dereference ptr to get the address of p1. Then, we use . operator to access the individual members.

We need the parenthesis to enclose \*ptr because the dot operator has higher precedence than the asterisk, resulting in an operation \*(ptr.price) which is definitely wrong because price is not a pointer.

Because the syntax ( \*ptr).price is very clumsy, C provides alternate syntax with -> (pointer operator ) like

ptr->price = 43.99

ptr->size = 32

***Please note there shouldn't be any space between – and >***

This pointer to structure and accessing members using -> is little tricky, so lot of practice is needed.

**Structure : Pass By Value and Pass by Reference**

We can pass structures to functions by value much similar to other variables. We do not pass the contents, but the copy of the variable. The invoked function can change the copy, but not the actual value of the variable that was passed.

Consider this example,

typedef struct \_profile {

char name [ 16 ] ;

int age ;

} Person ;

void printProfile ( struct \_profile p1 )

{

// a copy of the struct is passed in

p1.age ++ ; // no effect in the main function variable p.age

printf ( " age = %d name=%s \n" , p1.age, p1.name ) ;

}

main ( )   
{

Person p = { "Sam", 10 } ;

printProfile ( p ) ;

}

*If an array is present in the structure as a member, the array is also sent as value when the structure is passed to function.*

Structures can also be passed by reference.

**Array of Structures**

consider this simple int array

int data[3] = { 30, 40, 50 };

int \*ptr = data ; // make ptr point to data

We can print the values of data using the pointer ptr

for ( i = 0 ; i < 3 ; i++ ) {

printf ( " %d \n", \*ptr ) ;

ptr++ ; // advance the pointer to the next cell

}

*In the above code, \*ptr prints the cell data and ptr++ is advanced to the next cell because the ptr is type int and the array is also type int.*

We can define array of structures similarly.

We can define array of structures

struct \_data {

char name[12] ;

int age ;

} ;

struct \_data \*ptr , dataArray [ 3 ] =

{ "Carolyn", 30, "Barry", 50, "Pamela", 40 };

Here I defined an array of structure and Initialized them during declaration too.

NOTE HOW I defined them. You have to pay utmost care in making sure the order of the definition of the members.

Now let us point ptr to the first cell or the address of the first cell and first member

ptr = dataArray ;

// ptr = &dataArray[0] ; // alternate way

Generally the first method is preferred

// METHOD 1 , we use the pointer arithmetic to browse

// each and every cell

for ( i = 0 ; i < 3 ; i++ ) {

printf ( "%d %s \n", ptr->age, ptr->name);

ptr++;

}

// METHOD 2, we use pointer offset to navigate

ptr = dataArray ;

for ( i = 0 ; i < 3 ; i++ )

printf ( "%d %s \n", (ptr+i)->age, (ptr+i)->name) ;

// METHOD 3, we use pointer as an array

ptr = dataArray ;

for ( i = 0 ; i < 3 ; i++ )

printf ( "%d %s \n", ptr[ i ] .age, ptr [ i ] .name);

// METHOD 4, using the array itself

for ( i = 0 ; i < 3 ; i++ )

printf ( "%d %s \n",

dataArray[i].age, dataArray[i].name);

// METHOD 5 // not a elegant method

ptr = dataArray ;

for ( i = 0 ; i < 3 ; i++ )

printf ( "%d %s \n", (\*(ptr+i)) .age, (\*(ptr+i) ). name);

If you are using array, you could use Method 4. If you are using pointer, Method 1 can be used. They are the standard method of accessing members variables using pointers