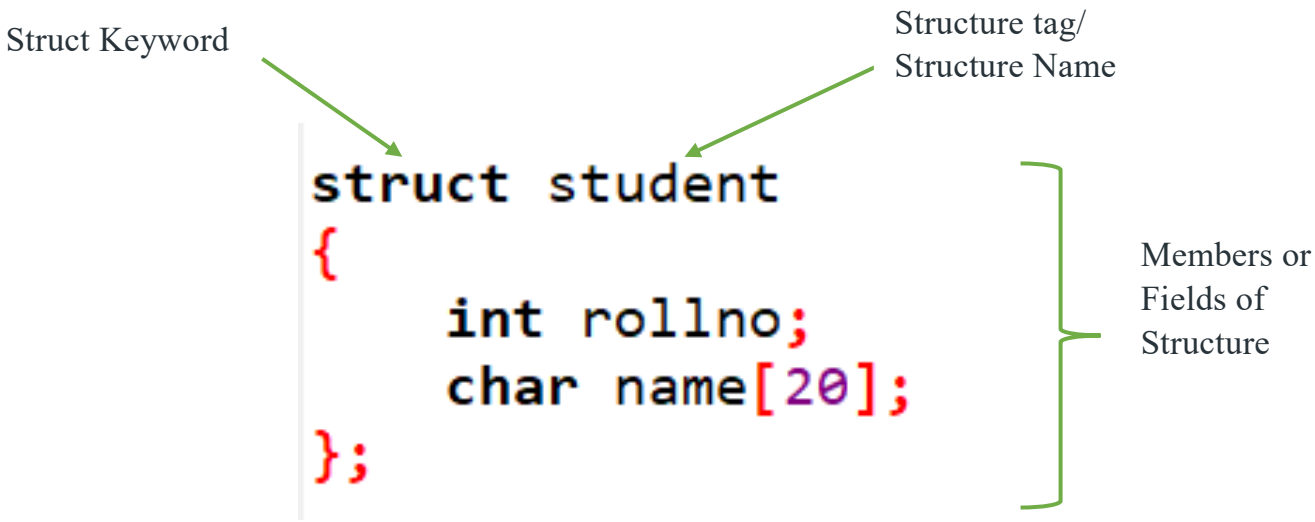
Structure

⇒ The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct keyword** is used to define the structure in the C programming language.

⇒ The items in the structure are called its **member** and they can be of any valid data type.



C Structure Declaration

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables a long with their datatype.

We can use the struct keyword to declare the structure in C using the following syntax:

Syntax

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
};
```

The above syntax is also called a prototype and no memory is allocated to the structure in the declaration.

structure template or structure

C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
}variable1, varaible2, ...;
```

2. Structure Variable Declaration after Structure Template

```
// structure declared beforehand
struct structure_name variable1, variable2, .......;
```

Access Structure Members

We can access structure members by using the **(.) dot operator.**

```
structure_name.member1;
strcuture_name.member2;
```

Initialize Structure Members

1. Initialization using Assignment Operator

```
struct structure_name str;
str.member1 = value1;
str.member2 = value2;
str.member3 = value3;
.
.
.
```

2. Initialization using Initializer List

```
struct structure_name str = { value1, value2, value3 };
```

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

```c
#include<stdio.h>

struct Person
{
   int id;
   char name[30];
   char city[20];
   float salary;
};

main()
{
   struct Person P;

   P.id=101;
   strcpy(P.name,"Rajesh Kumar");
   strcpy(P.city,"Ahmedabad");
   P.salary=25000;

   printf("\n\n Person : 1");
   printf("\n .............................\n");
   printf("\n Person Id     : %d",P.id);
   printf("\n\n Person Name   : %s",P.name);
   printf("\n\n Person City   : %s",P.city);
   printf("\n\n Person Salary : %.2f",P.salary);
}
```

⇒ The **typedef** is a keyword that is used to provide existing data types with a new name. The C typedef keyword is used to redefine the name of already existing data types.

⇒ When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands.

C typedef Syntax

```
typedef existing_name alias_name;
```

```c
#include<stdio.h>

typedef struct Person
{
   int id;
   char name[30];
   char city[20];
   float salary;
}Per;

main()
{
   Per p1;

   p1.id=101;
   strcpy(p1.name,"Rajesh Kumar");
   strcpy(p1.city,"Ahmedabad");
   p1.salary=25000;

   printf("\n\n Person : 1");
   printf("\n .............................\n");
   printf("\n Person Id     : %d",p1.id);
   printf("\n\n Person Name   : %s",p1.name);
   printf("\n\n Person City   : %s",p1.city);
   printf("\n\n Person Salary : %.2f",p1.salary);
}
```

Note : typedef is used to create alias for data types.

Structure Variable Array

```c
#include<stdio.h>

struct Employee
{
	int eid;
	char ename[30];
	float salary;
}E[3];

main()
{
	int i;

	for(i=0;i<3;i++)
	{
	    printf("\n\n .................Employee [%d]...........",i);
	    printf("\n\n Employee ID   [%d] : ",i);
	    scanf("%d",&E[i].eid);
	    printf("\n\n Employee's  Name      : ");
	    scanf("%s",&E[i].ename);
	    printf("\n\n Employee's  Salary    :  ");
	    scanf("%f",&E[i].salary);

	}


	for(i=0;i<3;i++)
	{
	    printf("\n\n .................Employee [%d]...........",i);
	    printf("\n\n Employee ID   [%d]   :  %d",E[i].eid);
	    printf("\n\n Employee's  Name      : %s",E[i].ename);
	    printf("\n\n Employee's  Salary    :  %.2f",E[i].salary);
	}
}
```

Nested Structure

A **nested structure** in C is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.

```c
#include<stdio.h>

struct Department    //Child structure
{
        int dept_id;
        char dept[30];
};

struct Employee     //Parent Structure
{
   int eid;
   char ename[30];
   float salary;

   struct Department Dept;  //Declaring structure variable for child class here.
};
```

```c
main()
{
   struct Employee Emp;

   Emp.eid=101;
   strcpy(Emp.ename,"Mr. A. A. Shukla");
   Emp.salary=33000;
   Emp.Dept.dept_id=1;
   strcpy(Emp.Dept.dept,"Purchase");

   printf("\n\n ..........Details From Employee Structure.............");
   printf("\n\n Employee's Id         : %d",Emp.eid);
   printf("\n\n Employee's Name       : %s",Emp.ename);
   printf("\n\n Employee's Salary     : %.2f",Emp.salary);

   printf("\n\n ..........Details From Department Structure.............");
   printf("\n\n Department's Id       : %d",Emp.Dept.dept_id);
   printf("\n\n Department's Name     : %s",Emp.Dept.dept);


}
```

```c
#include<stdio.h>

struct Department
{
        int dept_id;
        char dept[30];


        struct Employee
        {
            int eid;
            char ename[30];
            float salary;
        }Emp;

}Dept;
```

```c
main()
{

    Dept.Emp.eid=101;
    strcpy(Dept.Emp.ename,"Mr. A. A. Shukla");
    Dept.Emp.salary=33000;
    Dept.dept_id=1;
    strcpy(Dept.dept,"Purchase");

    printf("\n\n ..........Details From Employee Structure............");
    printf("\n\n Employee's Id        : %d",Dept.Emp.eid);
    printf("\n\n Employee's Name      : %s",Dept.Emp.ename);
    printf("\n\n Employee's Salary    : %.2f",Dept.Emp.salary);

    printf("\n\n ..........Details From Department Structure............");
    printf("\n\n Department's Id      : %d",Dept.dept_id);
    printf("\n\n Department's Name    : %s",Dept.dept);


}
```

Union

Union is an user defined datatype in C programming language. It is a collection of variables of different datatypes in the same memory location.

You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};
```

```c
int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Memory size occupied by data : 20
```

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Pointer

A **pointer** is a variable that **stores** the **memory address** of another variable as its value. A **pointer variable points** to a **data type** (like int) of the same type, and is created with the * operator. The address of the variable you are working with is assigned to the pointer:

```
int myAge = 43;     // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that stores the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Create a pointer variable with the name ptr, that **points to** an int variable (myAge). Note that the type of the pointer has to match the type of the variable you're working with (int in our example).

Use the & operator to store the memory address of the myAge variable, and assign it to the pointer.

Now, ptr holds the value of myAge's memory address.