UNIT – III

FUNCTIONS

Function:

A function is a group of statements that together perform a task. Every C program has at least one function. We can divide up our program code into separate functions. This concept is known as Modular Programming.

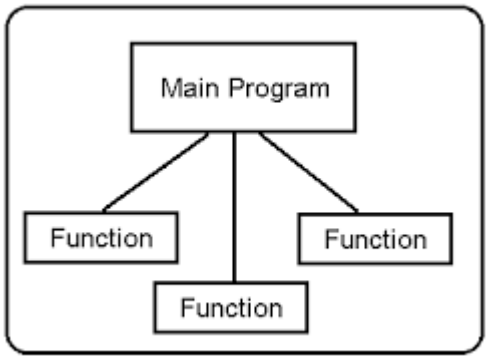


Fig. 1: Modular Programming

Need for User-Defined Functions in C:

Functions are used because of following reasons –

a) To improve the readability of code.

b)Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.

c)Debugging of the code would be easier if you use functions, as errors are easy to be traced.

d)Reduces the size of the code, duplicate set of statements are replaced by function calls.

The advantages of using functions are:

- Avoid repetition of codes.
- Increases program readability.
- Divide a complex problem into simpler ones.
- Reduces chances of error.
- Modifying a program becomes easier by using function.

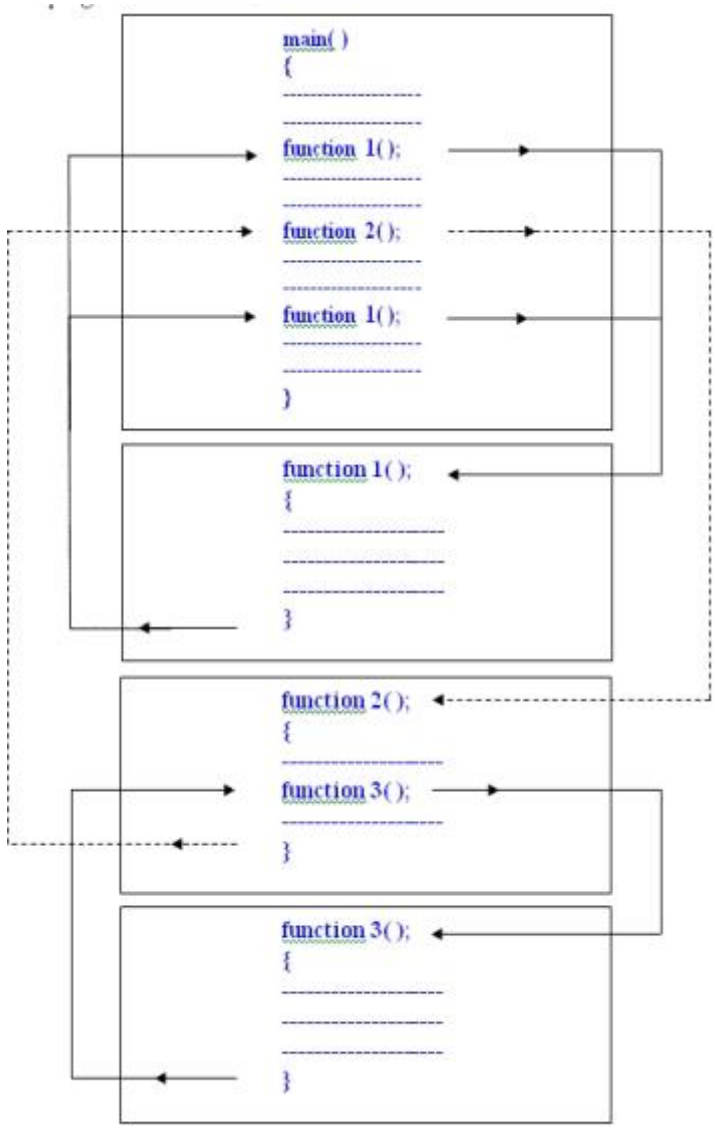


Figure : Flow of control in a multi-function program.

Types of the function:

There are two types of function.

1. Predefined Functions: The Predefined Functions are those which are already defined in the C Library / header files like printf(), scanf(), main() etc.

2. User defined Functions: These Functions are made by the user in the program itself to perform any task and solve any problem.

Elements of the User Defined Functions are broadly categorized into two parts.

1. Function header(Prototype)
2. Function Body(Definition)

Function declaration(Prototype) and Definition:

A function declaration tells the compiler about a function's name, return type, and parameters.

A function definition in C programming consists of a function header and a function body

The syntax of function can be divided into 3 aspects:

- Function Declaration
- Function Definition
- Function Call

General Format:

```
Return_type Functionname(parameter_list) // Function header
{
Statements; // Function Body
}
```

There are **5 categories of User defined functions**:

1) Functions with arguments and return value

2) Function with No arguments and a return value

3) Function with arguments and No return value

4) Function with No arguments and No return value

5) Function returning multiple values

1) Functions with No arguments and No return value:

Syntax:

```
Void functionname (void)
{
Statements; // Body of the function
}
```

 Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

 Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

 Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the data type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain **no parameters**

 Function Definition: function definition in which number of statements are written into body of the function.

 Function call: When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

Syntax for calling a function:

Functionname();

Example:

```
#include<stdio.h>
void sum(void); //Function Declaration
void sum(void) // function header
{
// function definition begins
int a,b,c;
printf("enter two number\n");
scanf("%d%d",&a,&b);
c=a+b;
printf("sum is %d",c);
} // Function Definition ends
void main()
{
sum(); //function call
}
```

Output

```
enter two number
10 20
sum is 30
```

2) Function with no arguments and a return value:

This type of function have no arguments with any return value that is the passed back to the main().

Syntax:

```
returnvalue functionname (void)
{
Statements; // Function Body
}
```

Example:

```
#include<stdio.h>
int sum(void); //function declaration
int sum(void)
{
int a,b,c;
printf("enter two number\n"); //function defination
scanf("%d%d",&a,&b);
c=a+b;
return(c);
}
void main()
{
int c;
c=sum(); //function call
printf("sum is %d",c);
}
```

Output :

```
enter two number
10 20
sum is 30
```

3) Function with arguments and No return value:

This type of function have arguments (values) which is passed from the main(), the arguments should have datatype and every argument separated by comma(,). It doesn't return any value to the main function .

Syntax:

```
void functionname (parameter list)
{
Statements; // Function Body
}
```

Example:

```
#include<stdio.h>
void sum(int a,int b); //function declaration
void sum(int a,int b)
{
int a,b,c;
c=a+b; //function defination
printf("sum is %d",c);
}
void main()
{
int a,b;
printf("enter two number\n");
scanf("%d%d",&a,&b);
sum(a,b); //function call
}
```

Output

```
enter two number
10 20
sum is 30
```

4) Function with arguments and with return value: This type of function have two way communication between user defined function and main(). The user define function accepted the data from main() and send back the result to main().

Syntax:

```
Returntype functionname (parameter list)
{
Statements; // Function body
}
```

Example:

```
#include<stdio.h>
int sum(int a,int b); //function declaration
int sum(int a,int b)
{  //function definition  begins
int a,b,c;
c=a+b;
return ( c );
} //function definition ends
void main()
{
int a,b, c;
printf("enter two number\n");
scanf("%d%d",&a,&b);
c=sum(a,b); //function call
printf("sum is %d",c);
}
```

Output

```
enter two number
10 20
sum is 30
```

What formal and actual arguments?

Formal Arguments :

The formal arguments are the arguments in the function declaration. The scope of formal arguments is local to the function definition in which they are used. They belong to the called function.

Actual arguments :

The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Example:

```
#include<stdio.h>
int sum(int a,int b); //function declaration
int sum(int a,int b) // a,b are formal arguments
{
```

```
int a,b,c;
c=a+b; //function definition
return ( c );
}
void main()
{
int a,b, c;
printf("enter two number\n");
scanf("%d%d",&a,&b);
c=sum(a,b); //function calling statement, a,b are Actual Arguments
printf("sum is %d",c);
}
```

5. Function returning multiple values:

Returning multiple values Using pointers: Pass the argument with their address and make changes in their value using pointer. So that the values get changed into the original argument.

We can return more than one values from a function by using the method called "call by address", or call by reference. In the invoker function, we will use two variables to store the results, and the function will take pointer type data. So we have to pass the address of the data.

Example:

```
// Program to find the greatest of two numbers
#include <stdio.h>
void compare(int a, int b, int* great, int* small)
{
   if (a > b) {
       // a is stored in the address pointed
      // by the pointer variable *great
      *great = a;
      *small = b;
   }
   else {
      *great = b;
      *small = a;
```

```
    }
}

int main()
{
    int great, small, x, y;
     printf("Enter two numbers: \n");
    scanf("%d%d", &x, &y);
     // The last two arguments are passed
    // by giving addresses of memory locations
    compare(x, y, &great, &small);
    printf("\nThe greater number is %d and the smaller number is %d", great, small);
    return 0;
}
```

PASSING PARAMETERS TO FUNCTIONS

When a function is called, the calling function has to pass some values to the called functions.

Functions can be invoked in two ways: **Call by Value** or **Call by Reference**.

The parameters passed to the function are called **actual parameters** whereas the parameters received by the function are called **formal parameters.**

There are two ways by which we can pass the parameters to the functions:

1. **Call by value:**
   - ✓ Here the values of the variables are passed by the calling function to the called function.
   - ✓ If any value of the parameter in the called function has to be modified the change will be reflected only in the called function.
   - ✓ This happens as all the changes are made on the copy of the variables and not on the actual ones.

2. Call by reference

   - ✓ Here, the address of the variables are passed by the calling function to the called function.
   - ✓ The address which is used inside the function is used to access the actual argument used in the call.

- ✓ If there are any changes made in the parameters, they affect the passed argument.
- ✓ For passing a value to the reference, the argument pointers are passed to the functions just like any other value.

Call by Value:

In call by value method of parameter passing, the values of actual parameters are copied to the function's formal parameters.

- There are two copies of parameters stored in different memory locations.
- One is the original copy and the other is the function copy.
- Any changes made inside functions are not reflected in the actual parameters of the caller.

In call by value, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

In call by value, actual arguments will remain safe, they cannot be modified accidentally.

Example using Call by Value :

```
#include <stdio.h>
void swapByValue(int a, int b)
{
int t;
t = a;
a = b;
b = t;
printf(" Values In user define a: %d, b: %d\n", a,b);
}
int main() /* Main function */
{
int a1 = 10;
int b1 = 20;
/* actual arguments will be as it is */
swapByValue(a1, b1);
printf ("Values In main a1: %d, b1: %d\n", a1,b1);
}
```

OUTPUT

======

Values In user define a: 20, b: 10

Values In main a1: 10, b1: 20

Thus actual values of a and b remain unchanged even after exchanging the values of x and y in the function.

Call by Reference : In call by reference method of parameter passing, the address of the actual parameters is passed to the function as the formal parameters. we use pointers to achieve call-by-reference.

- Both the actual and formal parameters refer to the same locations.
- Any changes made inside the function are actually reflected in the actual parameters of the caller.

In call by reference, to pass a variable n as a reference parameter, the programmer must pass a pointer to n instead of n itself. The formal parameter will be a pointer to the value of interest. The calling function will need to use & to compute the pointer of actual parameter.

Example using Call by Reference:

```
#include <stdio.h>
void swapByValue(int *a, int *b); /* Function Prototype */
void swapByValue(int *a, int * b)
{
int t;
t = *a;
*a = *b;
*b = t;
printf(" Values In user define *a: %d, *b: %d\n", *a,*b);
}
int main() /* Main function */
{
int a1 = 10;
int b1 = 20;
/* actual arguments will be as it is */
swapByValue(&a1,& b1);
```

```
printf("Values In main a1: %d, b1: %d\n", a1, b1);
}
```

OUTPUT

```
========
Values In user define a: 20, b: 10
Values In main a1: 10 , b1:20
```

Thus actual values of a and b get changed after exchanging values of x and y.

Difference between the Call by Value and Call by Reference:

The following table lists the differences between the call-by-value and call-by-reference methods of parameter passing.

Call By Value	Call By Reference
While calling a function, we pass the values of variables to it. Such functions are known as “Call By Values”.	While calling a function, instead of passing the values of variables, we pass the address of variables (location of variables) to the function known as “Call By References.
In this method, the value of each variable in the calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function is copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them.
In call-by-values, we cannot alter the values of actual variables through function calls.	In call by reference, we can alter the values of variables through function calls.
Values of variables are passed by the Simple technique.	Pointer variables are necessary to define to store the address values of variables.

Call By Value	Call By Reference
This method is preferred when we have to pass some small values that should not change.	This method is preferred when we have to pass a large amount of data to the function.
Call by value is considered safer as original data is preserved	Call by reference is risky as it allows direct modification in original data

Call by Value means passing values as copies to the function, so that the original data is preserved and any changes made inside the function are not reflected in the original data, whereas Call by Reference means passing references to the memory locations of variables (in C we pass pointers to achieve call by reference), hence changes made inside the function are directly modified in the original values.

Recursion

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

- In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
#include <stdio.h>
/* Function declaration */
#include<stdio.h>
 int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
   void main()
{
  int number,fact;
```

```
 printf("Enter a number: ");
 scanf("%d", &number);
  fact = factorial(number);
 printf("Factorial of %d is %d\n", number, fact);
 return 0;
}
```

Output:

Enter a Number

5

Factorial of 5 is : 120

Difference between Recursion and Iteration:

Iteration	Recursion
Allows the execution of a sequential set of statements repetitively using conditional loops.	A statement in the function's body calls the function itself.
There are loops with a control variable that need to be initialized, incremented or decremented and a **conditional control statement** that continuously gets checked for the termination of execution.	A recursive function must comprise of **at least one base case** i.e. a condition for termination of execution.
The value of the control variable continuously approaches the value in the **conditional statement**.	The function keeps on converging to the defined base case as it **continuously calls itself**.
A control variable stores the value, which is then updated, monitored, and compared with the conditional statement.	Stack memory is used to store the current state of the function.
Infinite loops keep utilizing CPU cycles until we stop their execution manually.	If there is **no base case defined**, recursion causes a stack overflow error.
The execution of iteration is comparatively **faster**.	The execution of recursion is comparatively **slower**.

Nesting of Functions:

A nested function is a function defined inside the definition of another function. It can be defined wherever a variable declaration is permitted, which allows nested functions within

another functions. Within the containing function, the nested function can be declared prior to being defined by using the auto keyword. Otherwise, a nested function has internal linkage.

A nested function can access all identifiers of the containing function that precede its definition.

A nested function must not be called after the containing function exits.

Example:

```
#include <stdio.h>
void outerFunction()
{
   int x = 10;
   void innerFunction()
{
      printf("Value of x from innerFunction: %d\n", x);
   }
   innerFunction();
}
int main()
{
   outerFunction();
   return 0;
}
```

Explanation:

outerFunction: This function contains a local variable x and defines a nested function innerFunction.

innerFunction: This function is defined within outerFunction and can access variables from the enclosing scope (like x).

Calling innerFunction: The outerFunction calls innerFunction, which prints the value of x.

UNIT – IV

STRUCTURES AND UNIONS

A structure is a user defined data type in C. A structure creates a data type that can be used to group items of possibly different types into a single type. 'struct' keyword is used to create a structure.

Declaration of Structure:

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

Syntax:

```
struct structurename
{
Datatype variablename1;
Datatype variablename2;
.
.
Datatype variablename n;
}
```

Example:

```
struct address
{
char name[50];
char street[100];
char city[50];
char state[20];
int pin;
};
```

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**

General Format:

struct structure_name variable1, variable2, ...;

Access Structure Members:

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

Syntax

structure_name.member1;

strcuture_name.member2;

Initialization of Structure Members:

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator:

Syntax:

Objectname.variablename=value;

str.member1 = value1;

str.member2 = value2;

str.member3 = value3;

Here, str is a structure variable(object) and member1,member2 and member3 are the structure members.

2. Initialization using Initializer List:

Syntax

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.

3. Initialization using Designated Initializer List:

Designated Initialization allows structure members to be initialized in any order.

struct structure_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };

Example

```
#include<stdio.h>
struct student
{
int rollno;
char name[20];
};
void main()
{
struct student obj; //declaring structure object
printf("enter rollno and name\n");
scanf("%d%s",&obj.rollno,&obj.name);
printf(" rollno is %d \n",obj.rollno);
printf(" name is %s \n",obj.name);
}
```

Output:

```
enter rollno and name
23 rahul
rollno is 23
name is rahul
```

Array of Structures:

- An array is a collection of elements of same data type that are stored in contiguous memory locations.
- A structure is a collection of members of different data types stored in contiguous memory locations.
- An array of structures is an array in which each element is a structure.
- This concept is very helpful in representing multiple records of a file, where each record is a collection ofdissimilar data items.
- As we have an array of integers, we can have an array of structures also.

Syntax:

struct structurename arrayname[size];

Let's take an example, to store the information of 3 students, we can have the following structure definition and declaration,

```
struct   student
{
            char    name[10];
            int     age;
            float   height;
};
```

```
struct student stu[3];
```

- Defines an array called stu, which contains three elements. Each element is defined to be of type structstudent.
- For the student details, array of structures can be initialized as follows,

```
struct  studentstu[3]={{"Dhars", 18 ,5.7},{"kani", 21, 5.8},{"Ramya",18, 5.5}};
```

ARRAYS WITHIN STRUCTURE

- It is also possible to declare an array as a member of structure, like declaring ordinary variables.

- For example to store marks of a student in three subjects then we can have the following definition of astructure.

```
struct   student
{
            char    name[10];
            int     rollno;
            int     marks[3];
};
struct student stu;
```

- Then the initialization of the array marks done as follows,

```
struct student stu= {"Deepika", 4 , {60,70,80}};
```

- The values of the member marks array are referred as follows,
  stu.marks [0] --> will refer the 0th element in the marks
  stu.marks [1] --> will refer the 1st element in the marks
  stu.marks [2] --> will refer the 2ndt element in the marks

NESTED STRUCTURE (STRUCTURE WITHIN STRUCTURE) :

A structure which includes another structure is called nested structure or structure within structure. i.e a structure can be used as a member of another structure.

There are two methods for declaration of nested structures.

(i) The syntax for the nesting of the structure is as follows

```
struct tag_name1
{
        type1 member1;
        .......
        .......
};
struct tag_name2
{
        type1 member1;
        ......
        ......
        struct tag_name1 var;
        ......
};
```

After declaring a structure like the above one, use the following **Syntax** to access the strcture variable.

outer_structure_variable.inner_structure_variable.membername;

ii) The syntax of another method for nesting of structures as follows

```
struct outer_struct
{
        datatype element-1;
        datatype element-2;
        ------------
        datatype element-n;
        struct inner_struct
        {
        datatype element-1;
        datatype element-2;
        ------------
        datatype element-n;
        }inner_struct_variable;
}outer_struct_variable;
```

Example for Nested structure:

```
struct stud_Res
 {
        int rno;
        char nm[50];
        char std[10];
        struct  stud_subj
        {
                char subjnm[30];
                int marks;
        }subj;
}result;
```

In above example, the structure stud_Res consists of stud_subj which itself is a structure with twomembers. Structure stud_Res is called as 'outer structure' while stud_subj is called as 'inner structure.'

The members which are inside the inner structure can be accessed as follow :

result.subj.marks=50;

STRUCTURES AND FUNCTIONS:

- Most C compilers, will allow you to pass entire structures as parameters and return entirestructures.
- As with all C parameters structures are passed by value and so if you want to allow a function toalter a parameter you have to remember to pass a ***pointer*** to a **struct**.

Example:

```
#include <stdio.h>
#include <string.h>
struct student
{
int id;
float percentage;
};
void func(struct student record);
void func(struct student record)
{
printf(" Id is: %d \n", record.id);
printf(" Percentage is: %f \n", record.percentage);
}
int main()
{
```

```
struct student record;
record.id=1;
record.percentage = 86.5;
func(record);
return 0;
}
```

Output

```
Id is: 1
Percentage is: 86.500000
```

Structure using Pointer:

Dot(.) operator is used to access the data using normal structure variable and arrow (->) is used to access the data using pointer variable.

Syntax:

```
Objectname->variablename=value;
```

Example:

```
#include<stdio.h>
#include <string.h>
struct student
{
int id;
char name[30];
float percentage;
};
int main()
{
int i;
struct student record1 = {1, "Raju", 90.5};
struct student *ptr;
ptr = &record1;
printf("Records of STUDENT1: \n");
printf(" Id is: %d \n", ptr->id);
printf(" Name is: %s \n", ptr->name);
printf(" Percentage is: %f \n\n", ptr->percentage);
return 0;
}
```

Output

Records of STUDENT1:

Id is: 1

Name is: Raju

Percentage is: 90.500000

Union:

Like Structures, union is a user defined data type. In union, all members share the same memory location. A union is a special data type that allows to store different data types in the same memory location.

We can define a union with many members, but only one member can contain a value at any given time

Size of a union is taken according the size of largest member in union. The keyword 'union' is used to declare the union in C.

Syntax:

```
union structurename
{
Datatype variablename1;
Datatype variablename2;
….
….
Datatype variablename n;
};
```

Example

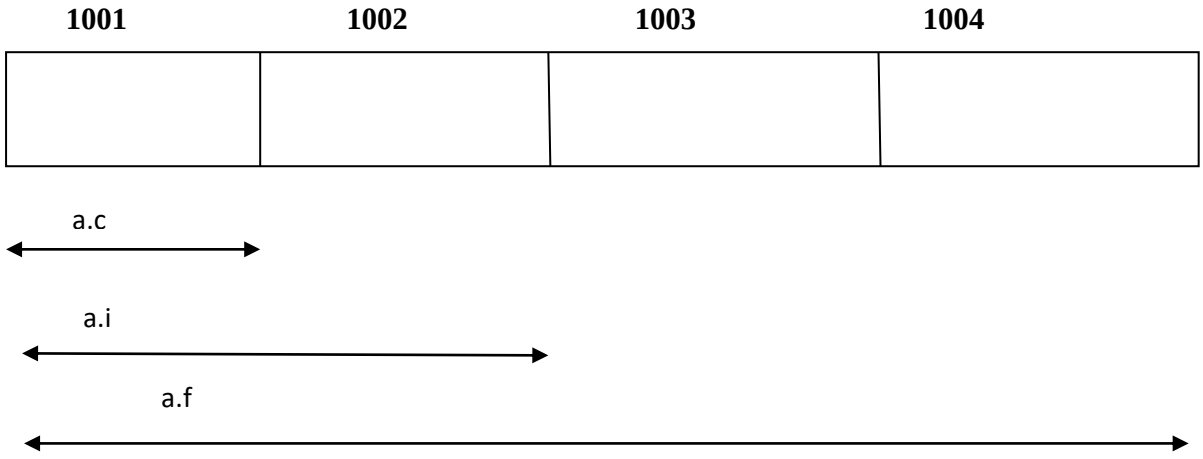
```
#include<stdio.h>
union student
{
int rollno;
char name[20];
};
void main()
{
union student obj; //declaring union object
printf("enter rollno and name\n");
scanf("%d%s",&obj.rollno,&obj.name);
```

```
printf(" rollno is %d \n",obj.rollno);
printf(" name is %s \n",obj.name);
}
```

A union definition and variable declaration can be done by using any one of the following:

union u
{
char c;
int i;
float f;
};
union u a;
	union u
	{
	char c;
	int i;
	float f;
	} a;
		typedefunion
		{
		char c;
		int i;
		float f;
		}U;
		U a;

We can access various members of the union as mentioned: a.c, a.i, a.f and memory organization is shown below,



a) Memory Organization union

- In the above declaration, the member f requires 4 bytes which is the largest among all the members.
- Figure shows how all the three variables share the same address. The size of the union here is 4 bytes.
- A union creates a storage location that can be used by any one of its members at a time.
- When a different member is assigned a new value, the new value supersedes the previous members'value.

==========