UNIT IV

User defined Functions

Introduction:

- A function is a self-contained program segment that carries out some specific, well defined task.
- Functions break large computing tasks into smaller ones.

Uses:

- Functions avoids redundant programming of the same instructions can be placed within a single function.
- It can be accesses whenever it is needed.
- Different set of data can be transferred to the function each time it is accessed.
- C supports the use of library functions, which are used to carry out a number of commonly used operations or calculations.
- C allows programmers to write their own functions.
- The same function can be accesses from different places within a program.

Example Program

```
# include <stdio.h>
main()
{
intaddnum(int, int);
intsum,a,b;
scanf("%d%d",&a,&b);
printf("\n The sum of %d and %d is %d",a,b,sum);
}
intaddnum(num1,num2)
{
int tot;
tot=num1+num2;
return(tot);
}
```

C functions can be classified into two:

- Library Functions
- User-Defined Functions

Ex: Printf, Scanf ,Sqrt,cosetc are Built-in functions or Library functions.

Library functions are not required to be written by us.

The user-defined function has to be developed by the user at the time of writing a program.

Need for User-Defined Functions:

- The program may become too large and complex and as result the task of debugging, testing and maintaining becomes difficult.
- Program is divided into functional parts, then each part may be independently coded and later combined into a single unit.
- These subprograms are called functions.

Needs:

- When certain type of operations or calculations is repeated at many points throughout a program.
- Length of a source program can be reduced.
- A function may be used by many other program.

Elements of User-Defined Functions:

- Functions are classified as one of the derived data types in C.
- Function names are also like as an identifier like variables, function have types.
- Each function must be declared and defined before they are used in a program.

Three elements of Functions:

1. Function definition
2. Function call
3. Function declaration

- Function definition is an independent program module.
- Use this function we need to invoke it at a required place in the program known as function call.
- A function that invokes another function is known as calling function.
- The program that calls the function referred to as calling program, function is referred to as called program.
- A function which is invoked by another function is known as called function.
- Calling program should declare before the definitions of the function known as function declaration or function prototype.

Definition of Function:

Function definition are grouped into two parts:

- Function header
- Function body

1. Function header consists of three elements.
2. Function name, function type, and list of parameters.
3. Function body consists of local variable declarations, function statements, and a return statements.

Syntax:

```
Function_typefunction_name(parameter_list)
{
        Local variable declaration;
```

```
Executable statement1;
Executable statement2;
        .
Executable statementn;
Return statement;}
```

- The function header should not end with a semicolon.
- The function body follows the function header and it is always enclosed in braces.
- The body is composed of declaration and statements.

Function_type (or) Return_type:

- Specifies the data type of the value returned by the function
- It may be any data type other then array and functions.
- If it is omitted, it is assumed as int.
- If function returns no value, void is used.

Function_name:

- It is an identifier
- Rules applied for function_name is same as identifier.
- Items within the parentheses are called parameters or arguments.

Parameter_List:

- It is also known as formal arguments or parameters.
- Zero or more arguments may be used.
- Each parameter must be preceded by its data type.
- More than one parameters must be separated by commas.
- Parameters are used to pass the values into the function.
- For parameterless function the void is placed within the parenthesis.

Return statement:

- It consists of the keyword return followed by an expression within the block and its returns a single value to the calling function.
- Parantheses around the expression are optional.
- A value or an expression is omitted , if the function returns no value.
- Multiple return statement are allowed.

Points to remember:

- A function cannot be defined more than once in a program.
- One function cannot be defined within another function definition.
- Function definition may appear in any order.

Example:

```
Float mul(float x,float y)
{
  Float result;
  Result=x*y;
Return(result);
```

}

Syntax for return Statement:

Return; (or) return(expression);

Function Calls:

- A function can be called by simply using the function name followed by a list of actual parameters.
- Arguments are enclosed in parentheses .
- Function call should be end with semicolon.

Example:

```
main()
{
int y;
y=mul(10,5);
printf("%d\n",y);
}
```

- The actual parameters must match the function's formal parameters in type, order and numbers.
- Multiple actual parameters must be separated by commas.

Different ways to call a Function:

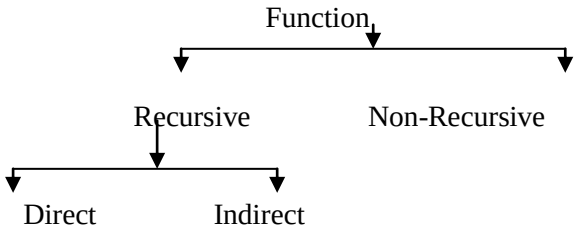
mul(10,5); mul(m,5); mul(5,n);

mul(m,n); mul(m+5,n) etc.

Syntax:

Function_name(v1,v2,…,vn);

- The formal arguments may be given the same names of the actual arguments or new names.
- If the formal and actual arguments disagree in their data types, or numbers of items.
- The garbage values are passed to the functions.
- Based on the function call, functions can be classified as



- Functions having recursive calls are known as recursive functions.
- Other than recursive types are known as non-recursive functions.
- If a function calls itself in the function body of its function definition, it is known as a direct recursive call.
- If a function calls another function, which in turn calls the first function, then it is called as an indirect recursive call.

Advantages of Recursion:

1. Easier understanding
2. Writing compact code
3. Easier coding

Disadvantage:

- Recursive function executes slowly.
- Additional storage space is required.

Function Declaration:

- Whenever a function is invoked in another function, it must be declared before use.
- Such a declaration is known as function declaration or function prototype.

Syntax:

return_typefunction_name(parameter_list);

- In the function declaration, parameter names are optional.
- It is possible to have the data type of each parameter.

  return_typefunction_name(data_type1,data_type2,….,data_typen);

- If the value returned by a function is int type, the declaration of function is optional.
- All other types of functions, declaration is mandatory.

Difference between function declaration and function definition.

Function Declaration	**Function Definition**
There is a semicolon at the end of parameter list	There is no semicolon at the end of parameter list.
The function body does not follow it.	The function body follows it.
Optional for Function returning int value	Mandatory for all function

1: No Arguments and No Return Values:

A function does not receive any data from the calling function.

- A function does not has arguments.
- The calling function does not receive any data from the called functions.
- There is no data transfer between the calling function and the called function.

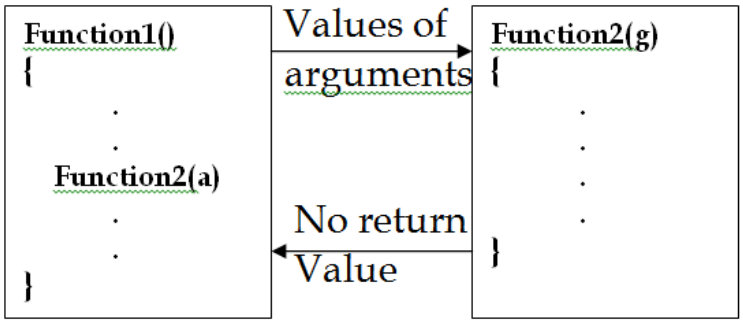
Example:

```
Void printline(void);
main()
{
printline();
}
Void printline(void)
{
int i;
```

```
for(i=1;i<=50;i++)
printf("%c",'-');
printf("\n");
}
```

Arguments but No returns Values:

- The main function has no control over the way the functions receive input data.
- The data communication between the calling function and the called function with arguments but no return value.



Syntax

```
main()
        Calling Function
{
….
function1(a1,a2);
  …..
}
function1(f1,f2)
{       f1,f2 are formal arguments.
 …..
……
}
```

- The actual and formal arguments should match in number, type and order.
- The actual arguments are more than the formal arguments, the extra arguments are discarded.
- If it is less that formal arguments the unmatched arguments are initialized to some garbage values.
- Function call is made, only a copy of the values of actual arguments is passed into the called function.

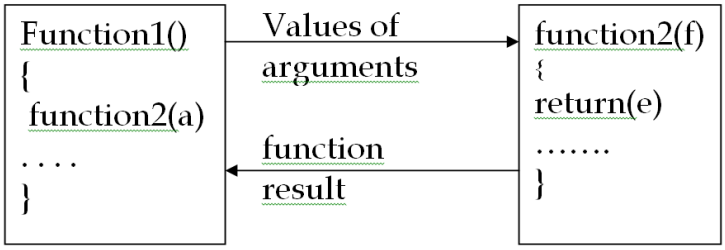
Example Program

```
Void printline(char c);
main()
{
Printline('z');
```

```
}
Void printline(char ch)
{
int i;
for(i=1;i<=50;i++)
  {
Printf("%c",ch);
Printf("\n");
}
```

Arguments with Return value:

- Two way data communication between calling and called functions.
- A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value.



Example Program

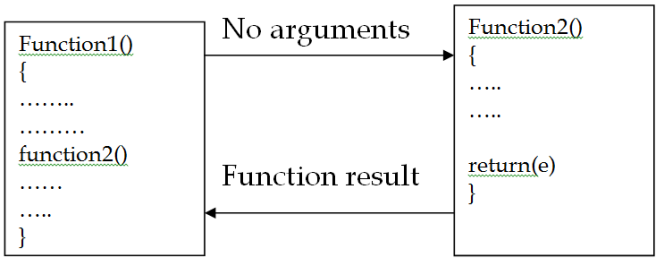
```
voidprintline(char ch);
int sum(int,int);
main()
{
inta,b,total;
scanf("%d%d",&a,&b);
printline('=');
total=sum(a,b);
```

```
voidprintline(char c)
{
int i;
for(i=1;i<30;i++)
printf("%c",c);
printf("\n");
}
int sum(intc,int d)
{
int tot;
tot=c+d;
return(tot);
}
```

ol along with a,b) to the formal arguments(c,d) are assigned the actu

2. The called function sum is executed line by line until the return(tot) statement encountered.
3. The integer value of sum is passed back to the function call in the function main.
4. The calling statement is executed and the returned value is assigned to total.
5. The next statement of the calling statement is executed until the end of main program.

No arguments but returns values:



- Function that may not take any arguments but returns a values to the calling function.

```
intget_number(void);
main()
{
int m=get_number();
printf("%d",m);
}
intget_number(void)
{
int number;
scanf("%d",&number);
return(number);
}
```

Functions that returns Multiple Values:

- A return statement can return only value.
- C using the arguments not only to receive information but also to send back information to the calling function.
- The arguments that are used to "send out" information are called output parameters.
- It can be achieved by using the address operator(&) and indirection operator(*).

Example Program

```
Void addsub(intx,inty,int *s,int *d);
main()
{
int x=20,y=10,s,d;
addsub(x,y,&s,&d);
printf("S=%d\n d=%d\n",s,d);
}
voidaddsub(inta,intb,int *sum,int *diff)
{
  *sum=a+b;
*diff=a-b;
}
```

- The variables *sum and *diff are known as pointer variables.

- The use of pointer variables as actual parameters for communicating data between functions is called “pass by pointers” or “call by reference”.

Points to remember:

- The type of actual and formal arguments must be same.
- The actual arguments must be the address of variables, the formal arguments must be prefixed by indirection operator *.

Nesting of Functions:

- C permits nesting of functions.
- Main can call function1, which calls function2, which calls function
- n3 and so on.,

```
float ratio(intx,inty,int z);
int difference(intx,int y);
main()
{
inta,b,c;
scanf(“%d%d%d”,&a,&b,&c);
printf(“%f\n”,ratio(a,b,c));
}
float ratio(intx,inty,int z)
{
if (difference(y,z))
return(x/(y-z));
else
return(0.0);
}
int difference(intp,int q)
{
if(p!=q)
return(1);
else
return(0);  }
```

Recursion:

- Recursion is a special case of this process.

A function calls itself.

```
main()
{
printf("This is a example of recursion \n");
main();
}
```

Syntax

This program will produce an indefinite output.

This is an example of recursion.

This is an example of recursion.

This is an example of recursion.

…..

…..

…..

The execution will continue indefinitely.

```
Factorial(int n)
{
int fact;
if(n==1)
return(1);
else
fact=n*factorial(n-1);
return(fact);
}
```

Passing Arrays to functions:

- C also permits to pass the values of an array to a function.
- To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments.

  Largest(a,n)

- Will pass the whole array a to the called function.
- Float largest(float array[],int size)
- Function largest is defined to take two arguments, the array name and the size of the array.

Example Program

```
main()
{   float largest(float a[],int n);
    float value[4]={2.5,1.5,3.2,4.2};
```

```
        printf("%f\n",largest(value,4));
}
float largest(float a[],int n)
{  int i;
float max;
max=a[0];
for(i=1;i<n;i++)
if(max<a[i])
max=a[i];
return(max); }
```

STRUCTURES AND UNIONS

Define structure:

- Structure is a derived data type to organize a group of related data items of different data types referring to a single entity.
- Single variable capable of holding data items of different data types.
- Structure variable is an object consisting of a sequence of named elements of various data types.
- Structures have declaration and definitions.
- The declaration of a structure does not reserve any storage space but the definition of a structure creates structures variables.

Syntax:

```
Struct   tag
{
declaration of member1;
declaration of member2;
…..
….
declaration of membern;
}
```

- The declaration begins with the keyword struct.
- The list of declarations of its member must be enclosed in braces.
- The tag is an identifier and it is optional.

Example:

```
Struct passenger
{
char name[30];
```

```
int age;
int train_no;
char destn[20];
char board_place[20];
}
```

- There are 5 members in this structure declaration.
- It describes the blueprint or template or shape of a structure.
- Tag is passenger.
- The tag name may be used as an ordinary variable name or a member variable name without any conflict.

Definition of structure variable:

```
Struct tag var1,var2,…..,varn;
```

For example:

```
Struct passenger pas1,pas2;
```

- Pas1 and pas2 are structure data type the structure definition reserve spaces for the structure variables.

Different ways to define structure variables:

Struct tag
{
member declarations;
};
struct tag var1,var2,….,varn;
	struct tag
	{
	member declarations;
	};
	typedef struct tag newname;
	newname var1,var2,…..,varn;

typedef struct { member declarations; } newname; newname var1,var2,….,varn;	struct tag { member declarations; } var1,var2,var3,…,varn;
struct { member declarations; } var1,var2,…,varn;	

- The blank space is optional after the closing braces in first, second, fourth and fifth method.
- But there should be atleast one blank spaces after closing braces in third method.
- Declaration part defines the template of the structure and it does not reserve the memory spaces.
- The structure variables are created and storage spaces are reserved.

INTIALISATION OF STRUCTURE VARIABLES:

- A structure variable can be initialized in its definition itself with a list of initializers enclosed within the braces.
- Each initialiser must be a constant expression and the order and type of each member must match the order and type of its declaration.

Example:

- Struct passenger pas1={"Priya",24,6500,"attur","salem"};
- Struct passenger pas2={"Uma",25,5200,"salem","Chennai"};

Example:

Structure passenger pas3=pas1;

Assigns the data of pas1 to pas3.

- But the structure variable at the right side of the assignment statement has to be defined and initialized or read before the assignment.

Example:

```
Struct employee
{
  int empno;
  char  e_name[20];
}emp1={2,"guna"},emp2={32="kalki"};
```

Accessing the members of a structure:

- Each member of a structure variable can be accessed using the structure member operator dot(.).

Syntax:

Structure_variable.member

Example:

- Members name and age of the structure passenger can be accessed as

```
Pas1.name /* refers to name of structure variable pas1 */
Pas2.age
```

- It is possible to assign values to all the individual members using the assignment statement.

```
Pas1.age=18;
Pas1.train_no=6020;
```

- The following assignment is incorrect since the member name is an array name which cannot be initialized.

```
Pas1.name="MALINI.R";   /* Invalid Assignment */
```

- The possible way assigning a string to a character array is

Form1:

```
Pas1.name[0]='M';
```

```
Pas1.name[1]='A';
Pas1.name[2]='L';
Pas1.name[3]='I';
Pas1.name[4]='N';
Pas1.name[5]='I';
Pas1.name[6]='\0';
```

Form 2:

Function scanf() may be used

```
Scanf("%s",pas1.name);
```

Form 3:

Function strcpy() may be used to copy a string

```
Strcpy(pas1.name,"MALINI");
```

Example program:

```
Main()
{
  typedef struct student
   {
     char name[20];
      int rollno;
       int mark;
   }STUDENT;
STUDENT s1,s2;   /* Structure definition */
Printf("Enter the values of each member in structure s1 \n");
Printf("Enter the name  :\n");
Scanf("%s",s1.name);
Printf("Enter the roll number:\n");
Scanf("%d",&s1.rollno);
Printf("Enter the marks :\n");
Scanf("%d",&s1.mark);
Printf("The members of structure S1:\n");
Printf("Name :%s\n",s1.name);
Printf("Roll number :%d\n",s1.rollno);
Printf(Mark : %d",s1.mark);
/* Assignment of one structure to another structure */
s2=s1;
printf("%s,%d,%d\n",s2.name,s2.rollno,s2.mark);
}
```

Nested structure:

- If a structure contains one or more structure as its members, it is known as a nested structure.

Example:

```
Structure date
{
  int day;
char month[10];
int year;
};
```

- The structure date may be used as a member in another structure as given below.

Syntax

```
Struct person /* Outer structure variable declaration */
{
   char name[30];
    int age;
     char sex;
    struct date dob;  /* Inner structure variable definition */
};
struct person man; /* Outer structure variable definition */
```

Example:

```
#include<stdio.h>
Main()
{
 typedef struct
 {
  Int date;
  Char month[10];
  Int year;
  }DATE;
Typedef struct
{
  Char name[25];
   Int age;
  Char sex;
  Date dob;
}person;
```

```
Printf("Enter the name, age and sex \n");
Scanf("%s%d%c",man.name,&man.age,&man.sex);
Printf("Enter the date of birth : date, month in words and year\n");
Scanf("%d%s%d",&man.dob.date,man.dob.month,&man.dob.year);
Printf("Enter the name, age and sex \n");
Printf("%s%d%c",man.name,man.age,man.sex);
Printf("Enter the date of birth : date, month in words and year\n");
printf("%d%s%d",man.dob.date,man.dob.month,man.dob.year);
}
```

Arrays of Structure:

- A group of structures may be organized in an array resulting in an array of structures.
- Each element in the array is a structure.

Example:

```
Struct person
{
 Char name[30];
 int age;
  char sex;
  struct date dob;
};
Struct person emp[10];
```

defines an array of 10 structures.

- Each structure variable emp[0],emp[1],…,emp[9] contains structure as its value.

The members are accessed as

emp[0].age	emp[0].sex	emp[0].dob.month
emp[1].age	emp[1].sex	emp[1].dob.month
.	.	.
.	.	.
.	.	.
emp[9].age	emp[9].sex	emp[9].dob.month

The character array can be accesses as

```
emp[0].name[0]='M';
emp[0].name[1]='A';
```

An array of structures can also be initialized similar to arrays.

```
Struct person emp[10]={{"Priya",8,'F'},{"Uma",10,'F'}};
```

Initializes two structure variables emp[0] and emp[1] partially.

Example Program

```
/* Program using an array of structures in functions */
```

```
#include<stdio.h>
typedef struct
{
  char name[10];
  int regno;
  char major[10];
  char result[10];
} STUDENT;

main()
{
  int n;
   STUDENT stud[10];
 Printf("Enter the number of student \n");
Scanf("%d",&n);
Readrecord(stud,n);
Writerecord(stud,n);
}
Readrecord(STUDENT stud[],int n)
{
  int i;
  for(i=0;i<n;i++)
  {
    Printf("Enter the name and register number of stud[%d]\n",i);
    Scanf("%s%d",stud[i].name,&stud[i].regno);
    Printf("Enter the major and result of stud[%d]\n");
    Scanf("%s%s",stud[i].major,stud[i].result);
}
}
Writerecord(STUDENT stud[],int n)
{
   int i;
   printf("the details of student records:\n");
   for(i=0;i<n;i++)
   {
    printf("%s%d",stud[i].name,stud[i].regno);
    printf("%s%s\n",stud[i].major,stud[i].result);
}
```

}

Pointer to structures:

- A pointer is helpful to create a structure dynamically. A group of structures may also be created dynamically using a pointer.
- A pointer to a structure is similar to a pointer to an ordinary variable.

Syntax

```
typedef struct stud
{
 int regno;
 char name[30];
 char major[10];
 char result[5];
}student;
                    Student *sp;
```

- After the creation of the pointer variable, it should be assigned with a suitable pointer value.

```
Sp=(student *)malloc(sizeof(int)+30+10+5;
```

- The operator sizeof may be used to find the size of a structure.

```
Sp=(student *)malloc(sizeof(student));
```

- To access the member of the structure variable as

(*sp).name and (*sp).regno may be used to access the first two members of the structure.

- The parentheses enclosing *sp are essential because the dot operator has higher precedence than the indirection operator.

```
*sp.regno /* is invalid */
```

- To avoid this confusion we can use another way to access the member as

```
Pointer to_structure->member name
```

- Now, (*sp).name can be written as sp->name and (*sp).regno can be written as sp->regno.

Example Program

```
/*Program to dynamically create a structure */
typedef struct
{
 char name[20];
 int empid;
}RECORD;
main()
{
```

```
 RECORD *r;
 r=(RECORD *)malloc(sizeof(RECORD));
 recordio(r);
printf("The given record :\n");
printf("%s%d\n",r->name,r->empid);
}
Recordio(RECORD *r)
{
 Printf("Enter the name and empid\n");
 Scanf("%s%d",r->name,&r->empid);
}
```

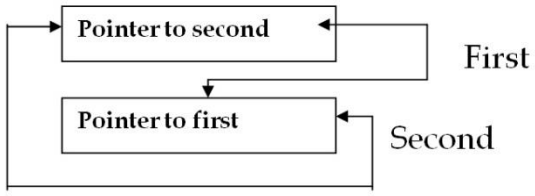
Self-Referential Structure:

- In a structure, if one or more members are pointer pointing to the same structure, then this structure is known as a self-referential structure.
- Structure itself as a member is not possible.
- There is also an indirect way of creating a self-referential structure as

Syntax

```
Struct first
{
        Struct second s;
};
Struct second
{
Struct first *f;
};
```

- The structure first has a member s pointing to the structure second, which has a member f pointing to the structure first.
- An indirect recursive declaration of a pointer to a structure also results in a self-referential structure.



UNIONS:

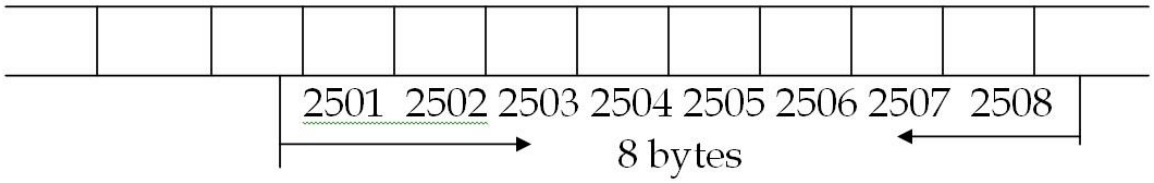
- A union can be considered as a special type of structure.

- It is a derived data types that permits different types of data items in which each member shared the same block of memory.
- The syntax for declaring, defining and accessing a union is borrowed from structures, except that the keyword union is used instead of struct.
- A union variable can be declared like a structure variable using the keyword union.

Declaration and definition of UNION:

```
Union mixed_type /* Union declaration */
{
        int i;
        float f;
        double d;
};
Union mixed_type mt; /* Union definition */
```

- The union variable mt contains three members mt.i, mt.f and mt.d.
- The variable mt will be large enough to hold the largest of the three data types.



- Only one of the members can be used at a time.
- The storage allocation used for each member is machine dependent.

```
Mt.i=50;
Mt.f=3.14;
```

- The value 3.14 is available in the same memory allocated for the union variable int. Hence the value 50 is lost and 3.14 is available.
- Initialization of a union variable is restricted so that the value of its first member can only be initialized.

```
Union mixed_type
{
float f;
int I;
} mix=5;
```

- A float value 5.0 would be stored in the memory mix.f;

Array and Structure

ARRAY	STRUCTURE
Array is a collection of data items of same data type	A structure is a collection of data items of different data types.
An array has declaration only	A structure may have a declaration followed by a definition
There is no keyword to mention the array data type	The keyword struct tells the complier that is a struct data type
An array declaration reserve enough memory space to store the value of its element	The definition tells the complier to reserve enough memory space
Array declaration allows initialization of the members	Structure definition only allowed to initialize the member values
An array cannot have bit fields	A structure may contain bit fields
An array declaration creates the array variable	A structure definition creates structure variables

Structure and Union

Structure	Union
Every memory has its own memory space.	Union use the same memory space to store the values
Keyword struct is used	Keyword union is used
Structure may be initialized with all its member	Only its first member may be initialized
Any member can be accessed at any time	Only one member can be handled at a time.
More storage space is required	Conservation of memory is possible.

BIT FIELDS:

- A word can be divided into a number of bit fields.
- A bit fields is one bit or a set of adjacent bits within a word.
- The size of a bit field varies from 1 to 16 bits in length.
- Direct manipulation of bit fields is allowed.

Syntax:

```
Struct   tagname
{
        Data_type mem_name1:field_width1;
        Data_type mem_name1:field_width2;
                        .
                        .
                        .
                        .
        Data_type mem_name1:field_widthn;
} var1,var2,…,varn;
```

- The data_type of the members can be int or signed int or unsigned int only.
- The fieldwidth specifies the number of bits used by the member.
- The bits are assigned left to right.

Member declared as

Data_type:fieldwidth;

Example:

```
Struct
        {
          int first_bit: 1  ;
               unsigned: 14;
         int last_bit: 1;
} bf;
```

It can be accessed by using structure variable.

```
bf.first_bit=1;
bf.last_bit=1;
```

- The values assigned are treated as integer values and hence they can be printed using %d in printf.
- Scanf() cannot be used to read a bit field.

Limitation of Bit Fields:

- Bit fields donot have addresses.
- Bit fields cannot be read using scanf() function.
- Bit field cannot be accessed using pointer
- Bit fields are not arrays.