UNIT – IV

CHAPTER IX

USER-DEFINED FUNCTIONS

9.1 INTRODUCTION

- C function can be classified into two categories, namely

(i) library functions

(ii) user-defined function.

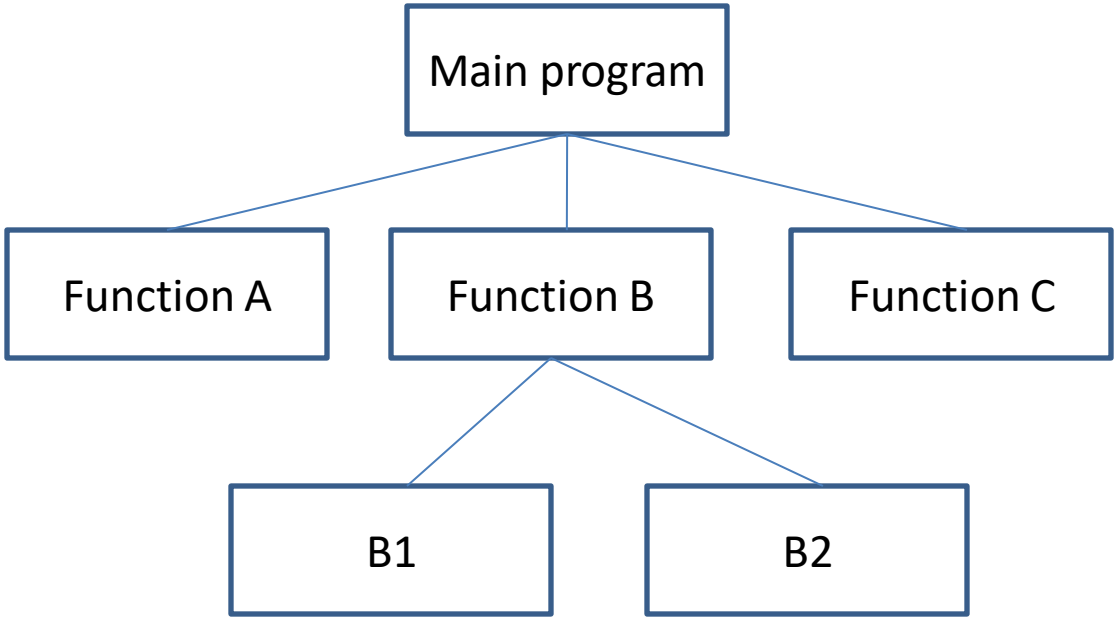
- **Main() is an example of user-defined functions.**
- While **printf()** and **scanf()** belong to the category of library functions**.**
- The main distinction between these two categories is that library functions are not required to be written by us whereas a user- defined function has to be developed by the user at the time of writing a program.

9.2 NEED FOR USER-DEFINED FUNCTION

- main() function indicates where the program has to begin its execution.
- While it is possible to code any program utilizing only main function but it leads to a lot of problem.
- Problem is that the program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult.
- If a program is divided into functional parts, then each part may be independently coded and later combined into single unit.
- These subprograms called 'function' are much easier to understand, debug and test.
- There are times when certain type of operation or calculation is repeated at many points throughout a program.
- So in such situation, we may repeat the program statements wherever they are needed.
- Another approach is to design a function that can be called and used whenever required.
- This saves both time and space.
- Top-down modular programming using function have the following advantage.

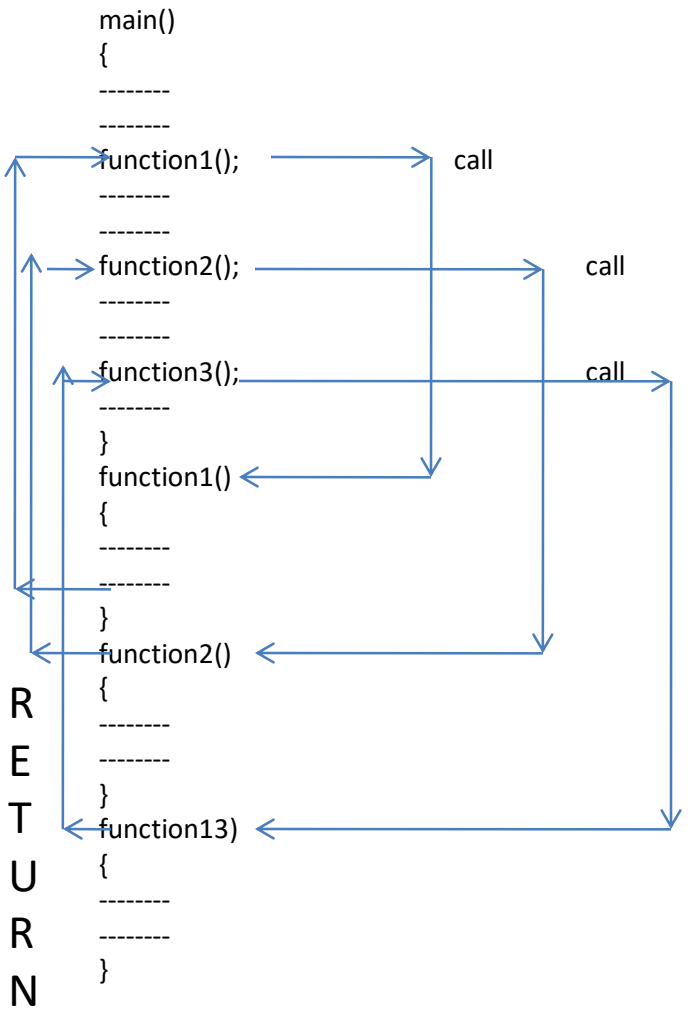
1. The length of a source program can be reduced by using function at appropriate places.
2. It is easy to locate and isolate a faulty function for further investigations.
3. A function may be used by many other programs.

Top-down modular programming using functions



9.3 A MULTIPLE-FUNCTION PROGRAM

- A function is a self-contained code that performs a particular task.
- Once a function has been designed , it can be treated as a block-box that takes some data from the main program and return a value.
- Every c program can be designed using the collection of these black boxes known as function.
- Any function can call any other function.
- It can call itself.
- A called function can also can call another function.
- A function can be called more than once.
- The function can be placed in any order.
- A called function can be placed either before or after the calling function.



9.4 MODULAR PROGRAMMING

- **Modular programming is a strategy applied to the design** and development of software systems.
- It is defined as organizing a large program into small, independent program segments called modules that are separately named and individually callable program units.
- These modules are carefully integrated to become a software system that satisfies the system requirements.
- It is basically a "divide-and-conquer" approach to problem solving.
- Modules are identified and designed such that they can be organized into a top-down hierarchical structure.

- Some characteristics of modular programming are:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.
5. All modules are designed as single-entry, single-exit systems using control structure.

9.5 ELEMENTS OF USER-DEFINED FUNCTION

- There are three elements of user-defined function.

1. Function Definition
2. Function Call
3. Function Declaration

- The **function definition** is an independent program module that is specially written to implement the requirement of the function
- In order to use this function we need to invoke it at a required place in the program. This is known as the **function call.**
- The program that calls the function is referred to as the **calling program or calling function.**
- The calling program should declare any function that is to be used later in the program.
- This is known as the **function declaration or function prototype.**

9.6 DEFINITION OF FUNCTIONS

- A function definition, also known as function implementation shall include the following elements.

1. Function name
2. Function type
3. List of parameters
4. Local variable declarations
5. Function statements
6. A return statement

- All the six elements are grouped into two parts, namely,

1. Function header (First three elements)
2. Function body (Second three elements)

a general format of a function definition to implement these two parts is given below.

```
function_type function_name(parameter list)
{
local variable declaration;
executable statement1;
executable statement2;
…….
…….
return statement;
}
```

- The first line function_type function_name(parameter list) is known as the function header and the statements within the opening and closing braces constitute the function body, which is a compound statement.

Function Header

- The function header consists of three parts: The function type(also known as return type, the function name and the formal parameter list. Note that a semicolon is not used at the end of the function header.

Name and Type

- The function type specifies the type of value that the function is expected to return to the program calling the function.
- If the return type is not explicitly specified, c will assume that it is an integer type.
- If the function is not returning anything, then we need to specify the return type as void.
- Remember, **void is one of the fundamental data types in C.**
- The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C.
- The name should be appropriate to the task performed by the function.

Formal Parameter List

- The parameter list declares the variables that will receive the data sent by the calling program.
- They serve as input data to the function to carry out the specified task.
- Since they represent actual input values, they are often referred to as formal parameters.
- These parameters can also be used to send values to the calling programs.
- These parameters are also known as arguments.

Examples

float quadratic(int a, int b, int c) { ….. }

- Remember, there is no semicolon after the closing parenthesis.
- Declaration of parameter variables cannot be combined.
- That is, int sum(int a,b) is illegal.
- A function need not always receive values from the calling program.
- In such cases, functions have no formal parameters.

- To indicate that the parameter list is empty, we use the keyword void between the parentheses as in

```
void printline(void)
{
.....
}
```

- This function neither receives any input values nor returns back any value.
- Many compilers accept an empty set of parentheses, without specifying anything as in

```
void printline()
```

- But, it is a good programming style to use void to indicate a null parameter list.

Function Body

- The function body contains the declarations and statement necessary for performing the required task.
- The body enclosed in braces, contains three parts, in the order given below:

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. **A return statement that returns the value evaluated by** the function.

- If a function does not return any value we can omit the return statement.
- However note that its return type should be specified as void.
- Again it is nice to have a return statement even for void functions.

Return Values And Their Types

- If a function return value then it is done through the return statement.
- While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most.
- The return statement can take one of the following forms

return;

or

return(expression);

- The first, the 'plain' return does not return any value; it acts much as the closing brace of the function.
- When a return is encountered, the control is immediately passed back to the calling function.

Example

```
int mul (int x, int y)
{
int result;
result =x*y;
return result;
}
```

- It return the value of p which is the product of the values of x and y, and the type of return value is int.

Function Calls

- A function can be called by simply using the function name followed by a list of actual parameters , if any, enclosed in parentheses.

Example

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

- When the compiler encounters a function call, the control is transferred to the function mul().
- This function is then executed line by line as described and a value is returned
- when a return statement is encountered.
- This value is assigned to y.

- A function call is a postfix expression. The operator (..)is at a very high level of precedence.
- Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.
- In a function call, the function name is the operand and the parentheses set which contains the actual parameters are the operator.
- The actual parameters must match the function's formal parameters in type, order and number.
- Multiple actual parameters must be separated by commas.

Note:

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

9.7 FUNCTION DECLARATION

A function declaration consists of four parts.

- Function type
- Function name
- Parameter list
- Terminating semicolon

Syntax

Function-type function-name (parameter list);

- The function declaration appear in the main function.
- Here function-type indicate the type of value that the function return.
- Function-name is name of the function.
- Parameter list is an arguments list.

Example

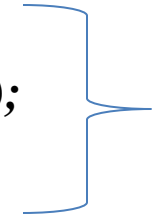
```
void main()
{
int sum(int a,int b);
--------
--------
}
int sum(int a, int b)
{
int c;
c=a+b;
return(c);
}
```

Points to note

1. The parameter list must be separated by commas.
2. The parameter names do not need not to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters then the list is written as void.
6. The return type is optional, when the functions returns int type data.
7. The return type must be void if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

Example

int mul(int, int);
int mul(int a, int b);
mul(int a, int b);
mul (int, int);



all are equally acceptable statements

Prototype : yes or no

- Prototype declarations are not essentials.
- If a function has not been declared before it is used, C will assume that its details available at the time of linking.
- Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definition.
- If these assumptions are wrong, the linker will fail and we will have to change the program.

Parameters Everywhere

Parameters (arguments0 are used in the following places:

1. in declaration (prototype)
   --- formal parameters
2. in function call
   ---actual Parameters
3. in function definition
   ---formal Parameters

- The actual Parameters used in calling statement may be simple constants, variables or expressions.
- The formal and actual Parameters must match exactly in type, order and number.
- Their names do not need to match.

9.8 CATEGORY OF FUNCTIONS

- A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

Category 1: Function with no arguments and no return Values.

Category 2: Function with arguments and return values.

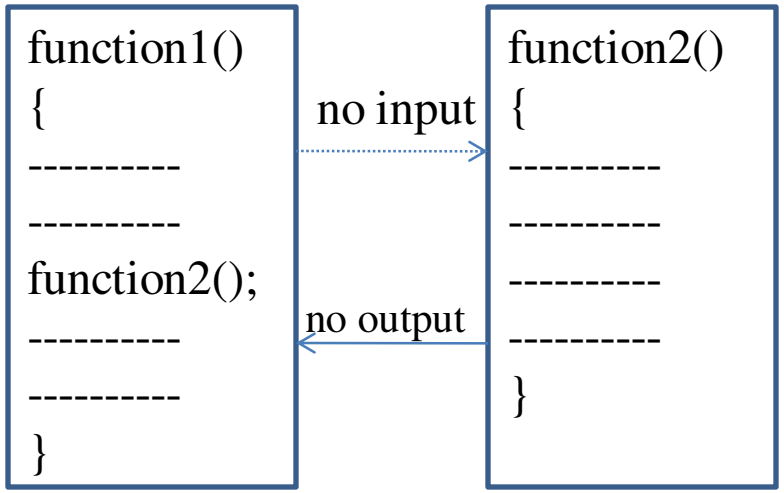
Category 3: Function with arguments and one return values.

Category 4: Function with no arguments and return values.

Category 5: Function that return multiple values.

9.8.1 NO ARGUMENTS AND NO RETURN VALUES

- When a function has no arguments, it does not receive any data from the calling function .
- Similarly, when it does not return value, the calling function does not receive any data from the called function.
- Ie, there is no data transfer between the calling function and the called function.



EXAMPLE 1

```
#include <stdio.h>
 void primeno();
 int main()
 {
primeno();// argument is not passed
return 0;
 }
 /* return type is void meaning
     doesn't return any value*/
 void primeno()
 {
 int n, i, flag = 0;
printf("Enter a positive integer value: ");
scanf("%d",&n);
 for(i=2; i <= n; ++i)
{
if( n%i == 0)
  {
     flag = 1;
  }
}
if (flag == 1)
     printf("%d is not a prime number.", n);
 else
     printf("%d is a prime number.", n);
 }
```

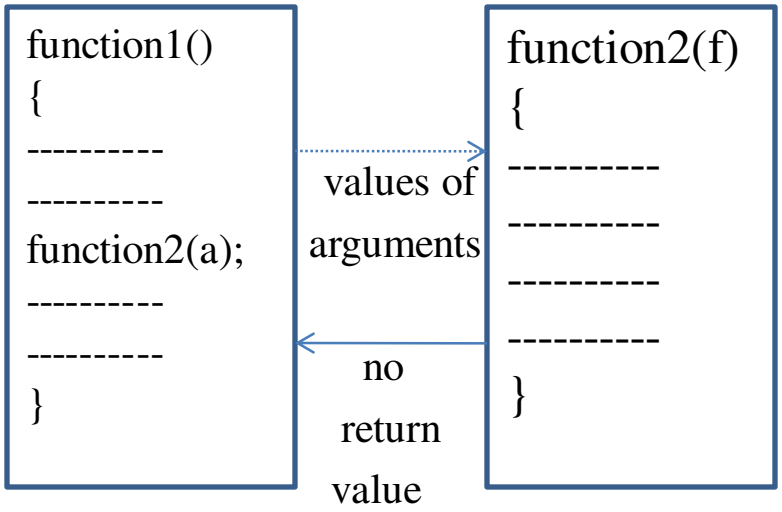
EXAMPLE 2

```
#include <stdio.h>
#include <conio.h>
void printline();
void si();
 int main()
 {
     clrscr();
     printline();
     si();
     printline();
     return 0;
 }
```

```
void printline()
{
     int i;
     for (i=0;i<=20;i++)
          printf("%c",'-');
     printf("\n");
}
void si()
{
     int p,n,r;
     float interest;
     printf("Enter principal, no. of years and
     rate of interest\n");
     scanf("%d%d%d",&p,&n,&r);
     interest=p*n*r/100.0;
     printf("Interest =%f",interest);
     getch();
}
```

9.8.2 Function with arguments and no return values

- The function that takes argument but no return value.
- The functions receives the arguments from the calling function but does not send back any value to calling function.



EXAMPLE

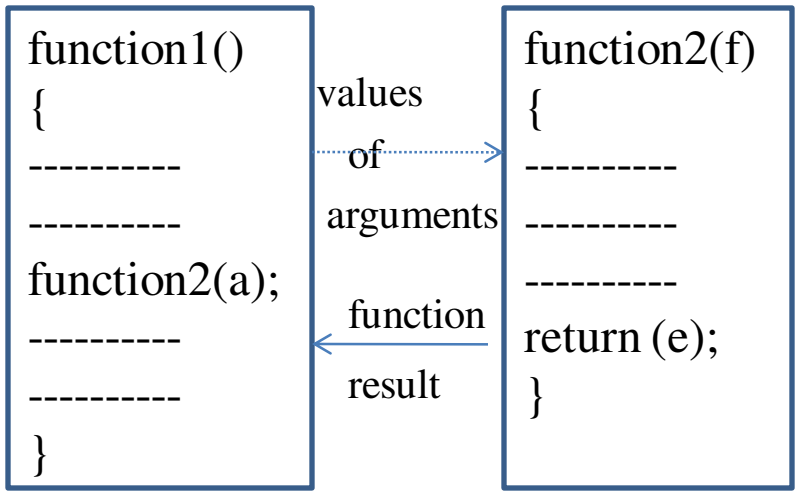
```
#include <stdio.h>
#include <conio.h>
void printline();
void si(int,int,int);
 void main()
 {
     int p1,n1,r1;
      clrscr();
     printf("Enter principal, no. of years
     and rate of interest\n");
     scanf("%d%d%d",&p1,&n1,&r1);
     printline();/*no arguments*/
     si(p1,n1,r1);/* with arguments*/
     printline();
     getch();
 }
```

```
void printline()
{
     int i;
     for (i=0;i<=20;i++)
              printf("%c",'-');
     printf("\n");
}

void si(int p,int n,int r)
{
     float interest;
     interest=p*n*r/100.0;
     printf("Interest=%f",interest);
}
```

9.8.3 FUNCTION WITH ARGUMENTS AND RETURN VALUES

- A self-contained and independent function should behave like a black box that receive a predefined form of input and outputs a desired value. Such function will have two-way data communication.
- The called function receives the arguments from the calling function and send back value to calling function.



EXAMPLE

```
#include <stdio.h>
#include <conio.h>
void printline();
float si(int,int,int);
 void main()
 {
     int p1,n1,r1;
     float intamt;
     clrscr();
     printf("Enter principal, no. of years
     and rate of interest\n");
     scanf("%d%d%d",&p,&n,&r);
     printline();/*no arguments*/
     intamt =si(p1,n1,r1);/* with
     arguments*/
     printf("simple interest= %f",intamt);
     printline();
     getch();
 }
```

```
void printline()
{
     int i;
     for (i=0;i<=20;i++)
          printf("%c",'-');
     printf("\n");
}

float si(int p,int n,int r)
{
     float interest;
     interest=p*n*r/100.0;
     return(interest);
}
```

The following events occur, in order, when the above function call is executed

1. The function call transfers the control along with copies of the values of the actual arguments to the function si where the formal parameters p, n and r are assigned the actual values of p1,n1 and r1 respectively.
2. The function si is executed line by line until the return(interest) statement is encounters. At this point, the float value of interest is passed back to the function call in the main and the following indirect assignment occurs:

   Si(int p1,int n1,int r1)=interest;
3. The calling statement is executed normally and the returned value is thus assigned to intamt, a float variable.

9.8.4 Function with no arguments and return values

- The called function does not receive any arguments but return value to the calling function.

Example

- getchar() function does not receive any arguments from the calling function but return a value.

Example

```
#include <stdio.h>
#include <conio.h>
int get_no();
```

```
void main()
{
    int n;
    n=get_no();
    printf("%d",n);
    getch();
}
int get_no()
{
    int no;
    scanf("%d",&no);
    return(no);
}
```

9.8.5 Function that return multiple values

- The mechanism of sending back information through arguments is achieved using what are known as the address operator (&) and indirection operator (*).
- 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.

Rules for pass by pointers.

1. The types of the actual and formal parameters must be same.
2. The actual arguments must be local and address variable.
3. The formal arguments in the function header must be prefixed by the indirection operator.
4. In the prototype, the arguments must be prefixed by the symbol *.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator.

Example

```
#include <stdio.h>
#include <conio.h>
void mathoperation(int ,int ,int *,int *);
void main()
{
int x=5, y=7,s,d;
clrscr();
mathoperation(x,y,&s,&d);
printf("s=%d\t d=%d",s,d);
}
void mathoperation(int a, int b, int *sum,  int *diff)
{
*sum=a+b;
*diff=a-b;
}
```

9.9 NESTING OF FUNCTIONS

- A called function can call another function.
- A function can call another function is called nesting of function.

Example

calculate the ratio a/(b-c)

```
float ratio(int x, int y, int z);
int diff(int x,int y);
void main()
{
int a,b,c;
clrscr();
printf("Enter the value of a, b and c\n");
scanf("%d%d%d", &a, &b, &c);
printf("Ratio = %f", ratio(a,b,c));
getch();
}
```

```
float ratio(int x, int y, int z)
{
if (diff(y,z))
   return(x/(y-z));
else
     return(0.0);
}

int diff(int y1,int z1)
{
if  (y1  != z1)
    return (1);
else
    return(0);
}
```

9.10 RECURSION

- When a called function in turn calls another function a process of "chaining" occurs. Recursion is a special case of this process, where a function calls itself.
- A very simple example of recursion

```
main()
{
        printf("this is an example of recursion");
        main();
}
```

- when executed, this program will produce an output something like this:

o/p: this is an example of recursion
this is an example of recursion
this is an example of recursion
this is an example of recursion
this is an example of

- Execution is terminated abruptly, otherwise the execution will continue indefinitely.

Example

```
/*Factorial of a given no
N!=n*(n-1)*(n-2)*…….*1  */
#include <stdio.h>
#include <conio.h>
int factorial(int no);
void main()
{
int n;
int fact;
clrscr();
printf("Enter the value of n\n");
scanf("%d",&n);
fact=factorial(n);
printf("Factorial of %d = %d",n,fact);
getch();
}
```

```
int factorial(int no)
{
    int f;
    if (no==1)
        return(1);
    else
        f=no*factorial(no-1);
    return (f);
}
```

9.11 PASSING ARRAYS TO FUNCTIONS

One-Dimensional Arrays

- Like the values of simple variables, it is also possible to pass the value of an array to a function.
- To pass a one-dimensional 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.

Example

largest(a,n);

- Will pass the whole array a to the called function.
- The called function expecting this call must be appropriately defined.
- The largest function header might look like:

float largest(float array[],int size)

- The function largest is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array.
- The declaration of the formal argument

array is

float array[];

- The pair of brackets informs the compiler that the argument ***array*** is an array of numbers.
- It is not necessary to specify the size of the ***array*** here.

```
#include <stdio.h>
void main()
{
float largest(float a[],int n);
float value[5]={1.1,3.7,-2.6,5.9,1.2};
printf("Largest =%f",largest(value,5));
}
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);
}
```

A[0] =1.1→max
A[1] = 3.7→max
A[2]= -2.6
A[3]= 5.9→max
A[4]= 1.2

- In C, the name of the array represent the address of its first element.
- By passing the array name, we are, in fact, passing the address of the array to the called function.
- The array in the called function now refers to the same array stored in the memory.
- Therefore, any changes in the array in the called function will be reflected in the original array.
- Passing addresses of parameters to the function is referred to as pass by address (or pass by pointers)

Three rules to pass an Array to a function:

1. **The function must be called by passing only the name of the array.**
2. **In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.**
3. **The function prototype must show that the argument is an array.**

Two-Dimensional Arrays

- Like simple arrays, we can also pass multi-dimensional arrays to functions.
- The approach is similar to the one we did with one-dimensional arrays.

The rules are simple.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

Example

```
#include <stdio.h>
#include<conio.h>
float average(int  x[][n],int m,int n);
void main()
{
 int m=3,n=3,i,j,a[3][3];
 float mean;
 clrscr();
 printf("enter the array elements one by one\m");
 for(i=0;i<m;i++)
   for(j=0;j<n;j++)
     scanf("%d",&a[i][j]);
 mean=average(a[][n],m,n);
 printf("average=  %f",mean);
 getch();
}
float average(int  x[][n],int m,int n)
{
 int i,j;
 float sum=0.0;
 for(i=0;i<m;i++)
      for(j=0;j<n;j++)
           sum=sum+x[i][j];
  return(sum/(m*n));
}
```

Passing Strings to Functions

The strings are treated as character arrays in C and therefore the rules for passing strings to function are very similar to those for passing arrays to the functions.

1. The string to be passed must be declared as a formal argument of the function when it is defined.
2. The function prototype must show that the argument is a string.
3. A call to the function must have a string array name without subscripts as its actual argument.

```
#include <stdio.h>
void  passstring (char str[]);
char s1[50];
int main()
 {
 char s[50];
printf(“Enter string: ");
gets(s);
/* passing string to a function.*/
passstring(s);
return 0;
 }
void passstring (char str[])
 {
printf(“String output: ");
puts(str);
}
```

Call by Value and Call by reference

- The technique used to pass data from one function to another is known as parameter passing.

Parameter passing can be done in two ways.

1. pass by value (also known as call by value)
2. pass by pointers (also known as call by reference)

1. Pass by value (call **by value)**

In pass by value,

- The value of actual parameters are copied to the variables in the parameter list of the called function.
- The called function works on the copy and not on the original values of the actual parameters.
- This ensures that the original data in the calling function cannot be changed accidentally.

*2. Pass by pointers (***Call by reference)**

- **The memory addresses of the** variables rather than the copies of values are sent to the called function.
- In this case, the called function directly works on the data in the calling function and the changed value is available in the calling function for its use.
- pass by pointers method is often used when manipulating arrays and strings.
- This method is also used when we require multiple values to be returned by the called function.

9.12 Scope, Visibility and Lifetime of Variables

Storage classes are

1. ***Automatic variables***
2. ***External variables***
3. ***Static variables***
4. ***Register variables***

- The scope of the variable determines over what region of the program a variable is actually available for use (‘active’)
- Longevity refers to the period during which a variable retains a given value during execution of a program (‘alive’).
- So longevity has a direct effect on the utility of a given variable.
- The visibility refers to the accessibility of a variable from the memory.
- The variables may also be categorized, depending on the place of their declaration, as

Internal (local) or

External(global)

- Internal (local) variables are those which are declared within a particular function.
- External variables are declared outside of any function.

Automatic variables

- Automatic variables are declared inside a function in which they are to be utilized.
- They are created when the function is called and destroyed automatically when the function is exited.
- Automatic variables are private variable to the function in which they are declared .
- Automatic variables are also called as local or internal variables.
- A variable declared inside a function without storage class specification is, by default.

Example

```
#include <stdio.h>
#include<conio.h>
void function1();
void function2();
void main()
{
int x=100;
function2();
printf("The  value of x inside main =%d",x);
getch();
}
void function1()
{
int x=10;
printf("The value of x inside function1() =%d",x);
}
void function2()
{
int x=1;
function1();
printf("The value of x inside function2() =%d",x);
}
```

OUTPUT

The value of x inside function1() =10

The value of x inside function2() =1

The Value of x inside main = 100

External variables

- Variables that are both alive and active throughout the program are known as external variables.
- They are also kown as global variables.
- Unlike local variables, global variables can be accessed by any function in the program.
- External variables are declared outside a function.

Example

```
int no;
float l=3.2;
main()
{
---------
---------
}
function1()
{
---------
---------
}
function2()
{
---------
---------
}
```

- The variable no and l are available for use in all the three functions.
- In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared.

Example

```
int count;
main()
{
count=10;
---------
}
function1()
{
int count=0
---------
}
```

- When the function1() references the variable count, it will referencing its local variable count(=0), not the global one.

Example program

```
#include <stdio.h>
#include<conio.h>
int f1(void);
int f2(void);
int f3(void);
int x;
void main()
{
x=100;
clrscr();
printf("x=%d\n",x);
printf("x=%d\n",f1());
printf("x=%d\n",f2());
printf("x=%d\n",f3());
getch();
}
f1()
{
x=x*10;
return(x);
}
```

```
f1()
{
int x;
x=10;
return(x);
}
f3()
{
x=x+10;
return(x);
}
```

Output

x=100

x=1000

x=10 → Local variable

x=1010

- Once a variable has been declared as global, any function can use it and change the value .
- Then subsequent functions can reference only that new value.

Global variable as parameters

- Using global variables as parameters for passing values produces problems.
- The values of global variables which are sent to the called function may be changed without knowledge by the called function.
- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

External declaration

- This tells External variables are also known as global variables.
- These variables are defined outside the function.
- These variables are available globally throughout the function execution.
- The value of global variables can be modified by the functions.
- "extern" keyword is used to declare and define the external variables.

Scope − They are not bound by any function. They are everywhere in the program i.e. global.

Default value − Default initialized value of global variables are Zero.

Lifetime − Till the end of the execution of the program.

1. External variables can be declared number of times but defined only once.
2. “extern” keyword is used to extend the visibility of function or variable.
3. By default the functions are visible throughout the program, there is no need to declare or define extern functions. It just increase the redundancy.
4. Variables with “extern” keyword are only declared not defined.
5. Initialization of extern variable is considered as the definition of the extern variable.

```
#include <stdio.h>
extern int x = 32;
int b = 8;
int main()
 {
  auto int a = 28;
extern int b;
printf("The value of auto variable : %d\n", a);
 printf("The value of extern variables x and b :
      %d,%d\n",x,b);
 x = 15;
 printf("The value of modified extern variable x :
      %d\n",x);
   return 0;
}
```

Output

```
The value of auto variable : 28
 The value of extern variables x and b : 32,8
 The value of modified extern variable x : 15
```

Consider the program segment

```
main()
{
y=5;
--------
--------
}
int y;
f1()
{
--------
y=y+1;
--------
}
```

- As for as main is concerned ,y is not defined.
- So, the compiler will issue an error message.
- Unlike local variables, global variables are initialized to zero by default.
- So the value of y in f1 is assigned to 1.

External declaration

- The above problem can be solved by declaring the variable with the storage class extern.

Example

```
main()
{
extern int y;
--------
--------
}
f1()
{
extern int y;
--------
--------
}
```

- The external declaration of y inside the function informs to the compiler that y is an integer type defined somewhere else in the program

Static variables

- Static variables are initialized only once.
- The compiler persists with the variable till the end of the program.
- Static variables can be defined inside or outside the function.
- They are local to the block.
- The default value of static variables is zero.
- The static variables are alive till the execution of the program.

Syntax

static datatype variable_name = value;

- **datatype** − The datatype of variable like int, char, float etc.
- **variable_name** − This is the name of variable given by user.
- **value** − Any value to initialize the variable. By default, it is zero.

Example1

```
#include <stdio.h>
 int main()
 {
auto int a = -28;
static int b = 8;
printf("The value of auto variable : %d\n", a);
printf("The value of static variable b:%d\n",
     b);
if(a!=0)
     printf("The sum of static variable and auto
     variable : %d\n",(b+a));
 return 0;
}
```

Output

The value of auto variable : -28

The value of static variable b : 8

The sum of static variable and auto variable : -20

Example 2

```
#include<stdio.h>
#include<conio.h>
void stat(void);
void main()
{
int i;
for (i=1;i<=3;i++)
   stat();
getch();
}
void stat(void)
{
static int x = 10;
x=x+1;
printf(“”x=%d\n”,x);
}
```

Output

x=11

x=12

x=13

- An external static variable is declared outside of all functions and is available to all the functions in that program.
- The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

Register variables

- Register variables tell the compiler to store the variable in CPU register instead of memory.
- Frequently used variables are kept in registers and they have faster accessibility.
- We can never get the addresses of these variables.
- "register" keyword is used to declare the register variables.
- **Scope** − They are local to the function.
- **Default value** − Default initialized value is the garbage value.
- **Lifetime** − Till the end of the execution of the block in which it is defined.

Example

```
#include <stdio.h>
int main()
 {
 register char x = 's';
 register int a = 10;
 auto int b = 8;
printf("The value of register variable b : %c\n",x);
printf("The sum of auto and register variable : %d",(a+b));
 return 0;
}
```

Output

```
The value of register variable b : S
The sum of auto and register variable : 18
```

- Register keyword can be used with pointer also.
- It can have address of memory location.
- It will not create any error.

Example

```
#include<stdio.h>
int main()
 {
   int i = 10;
   register int *a = &i;
   printf("The value of pointer : %d", *a);
   getch();
   return 0;
}
```

Output

```
The value of pointer : 10
```

Storage class	Where declared	Visibility	lifetime
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with extern	Entire program (global)
Extern	Before all functions in a file (cannot be initialized) extern and the file where originally declared as global	Entire file plus other files where variable is declared	global
Static	Before all functions in a file	Only in that file	global
None or auto	Inside a function (or a block)	Only in that function or a block	Until end of function or block
register	Inside a function (or a block)	Only in that function or a block	Until end of function or block
Static	Inside a function	Only in that function	global

UNIT – IV

CHAPTER X

STRUCTURES AND UNIONS

10.1 INTRODUCTION

- Arrays can be used to represent a group of data items that are belongs to the same type.
- Arrays are used to store large set of data and manipulate them.
- disadvantage of arrays is that all the elements stored in an array are to be of the same data type.
- If we need to use a collection of different data type items it is not possible using an array.
- When we require using a collection of different data items of different data types we can use a structure.
- **Structure is a method of packing data of different types.**
- A structure is a convenient method of handling a group of related data items of different data types.

10.2 Array vs structures

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas Structure is user-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a Structure, first, we have to design and declare a data structure before the variables of that type are declared and used.

10.3 DEFINING A STRUCTURE

Syntax

struct tag_name
{
data type member1;
data type member2;
…
…
};

Note

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name can be used to declare structure variables of its type, later in the program.

struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
};

- Create a structure Book_bank to store book details.
- The keyword struct declares a structure to holds the details of four fields namely title, author, pages and price.
- These are members or elements of the structures.
- Each member may belong to different or same data type.
- The tag name can be used to define objects that have the tag names structure.
- The structure we just declared is not a variable by itself but a template for the structure.

10.4 DECLARING STRUCTURE VARIABLE

To access structure item, we require to create

structure variable (object).

- A structure variable declaration is similar to the declaration of variables of any other data types.

It includes the following elements:

1. The keyword struct.
2. The structure tag name.
3. List of variable names separated by names.
4. A terminating semicolon.

Syntax

struct structure_name variable_name;

We can declare structure variables using the tag name any where in the program.

Example

struct lib_books book1,book2,book3;

- Declares book1,book2,book3 as variables of type struct lib_books each declaration has four elements of the structure lib_books.
- Structures do not occupy any memory until it is associated with the structure variable such as book1.

We can also combine both template declaration and variables declaration in one statement.

Example

```
struct lib_books
{
char title[20];
char author[15];
int pages;
float price;
} book1,book2,book3;
```

The use of tag name is optional.

Example

```
struct
{
…
…
…
} book1, book2, book3 ;
```

- Declares book1, book2, book3 as structure variables representing 3 books but does not include a tag name for use in the declaration.
- This approach is not recommended for the two reasons.

1. Without tag name, we cannot use it for future declarations.
2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the main, along with macro definitions, such as #define. In such cases, the definition is global and can be used by other functions as well.

10.5 Type-Defined Structures

Syntax

typedef struct

{

----

type member1;

type member2;

}type name;

- The type-name represents structure definition associated with it and therefore, can be used to declare structure variables as

type-name variable1,variable2,……..;

Note

1. The name type-name is the type definition name ,not a variable.
2. We cannot define a variable with typedef declaration.

10.6 ACCESSING STRUCTURE MEMBERS

- The link between a member and a variable is established using the member selection operator '.' which is known as **dot operator or period operator.**

Example

book1.price

- is the variable representing the price of book1 and can be treated like any other ordinary variable.
- we can assign variables to the members of book1

strcpy(book1.title,"basic");

strcpy(book1.author,"Balagurusamy");

book1.pages=250;

book1.price=285.0;

Or We can use scanf statement to assign values like

scanf("%s",book1.title);

scanf("%d",&book1.pages);

Example program

```
#include<stdio.h>
#include<conio.h>
struct student
{
char name[20];
int num;
int mark;
};
void main()
{
struct student s1,s2;
clrscr();
printf("Enter the name of student1:");
scanf("%s",s1.name);
printf("Enter the roll number of
    student1:");
scanf("%d",&s1.num);
printf("Enter the marks of student1:");
scanf("%d",&s1.mark);
printf("Enter the name of student2:");
scanf("%s",s2.name);
printf("Enter the roll number of
student2:");
scanf("%d",&s2.num);
printf("Enter the marks of student2:");
scanf("%d",&s2.mark);
printf("Students Details\n");
printf("Student 1 details\n");
printf("%s\n",s1.name);
printf("%d\n",s1.num);
printf("%d\n",s1.mark);
printf("Student 2 details\n");
printf("%s\n",s2.name);
printf("%d\n",s2.num);
printf("%d",s2.mark);
getch();
}
```

10.7 STRUCTURE INITIALIZATION

- A structure variable can be initialized at compile time.

Example 1

```
main()
{
struct
{
int weight;
float height;
} student={50,170.26};
.....
.....
}
```

- This assigns 50 to student.weight and 170.26 to student.height.
- There is one-to-one correspondence between the members and their initializing values.

Example2

```
main()
{
struct st_record
{
int weight;
float height;
};
struct st_record student1={50,170.26};
struct st_record student2={60,180.75};
.....
.....
}
```

Example 3

```
struct st_record
{
int weight;
float height;
}student1={50,170.26};

main()
{
struct st_record student2={60,180.75};
……
……
}
```

- The c language does not permit the initialization of individual structure members within the templates.
- The initialization must done only in the declaration of the actual variables.
- The compile-time initialization of a structure variable must have the following elements.

1. The keyword struct
2. The structure tag_name.
1. The name of the variable to be declared.
2. The assignment operator =.
3. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
4. A terminating semicolon.

Rules for initializing structures

1. We cannot initialize the individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values.

10.8 COPYING & COMPARING STRUCTURE VARIABLES

copy

- Two variables of the same structure type can be copied .

Example

- If student1 and student2 belong to the same structure then

  student1=student2;

  student2= student1;

Comparision

- The following statements are not permitted in c.

  student1 == student2

  student1 != student2

- C does not permitted any logical operators on structure variables.
- We can compare members individually.

```
structure class
{
int no;
char name[20];
int marks;
};
```

```
void main()
{
int x;
struct class stud1={101, "anu",89};
struct class stud2={102, "banu",80};
struct class stud3;
stud3=stud2;
x= ((stud3.no==stud2.no) && (stud3.marks ==
      stud2.marks))?1:0;
if (x=1)
{
printf("\n student2 and student3 same\n");
printf("%d%s%d",stud3.no,stud3.name,stud3.marks)
      ;
}
else
{
printf("student2 and student3 are different");
}
}
```

10.9 ARRAYS OF STRUCTURES

- Declaring an array of structure is same as declaring an array of fundamental types.
- An array is a collection of elements of the same type.
- In an array of structures, each element of an array is of the structure type.
- An array of structure is stored inside the memory in the same way as a multi-dimensional array.

Syntax

struct tag_name
{
data type member1;
data type member2;
…
…
};
struct tag-name struct_array[size];

Example

struct student
{
char name[20];
int num;
int mark;
};
struct student stud[10];

EXAMPLE

```
#include<stdio.h>
#include<conio.h>
struct student
{
char name[20];
int num;
int mark;
};
struct student stud[10];
void main()
{
int i,n;
clrscr();
printf("\n Enter the no. of students in a class:");
scanf("%d", &n);
printf("Enter the students detials one by one\n");
for (i=0;i<n;i++)
{
printf("Enter the reg.no of the student %d :", i+1)
scanf("%d", &stud[i].num);
printf("Enter the name of the student %d :", i+1)
scanf("%s", stud[i].name);
printf("Enter the marks of the student %d :", i+1)
scanf("%d", &stud[i].mark);
}
for (i=0;i<n;i++)
{
printf("Reg.no of the student %d : %d",
      i+1,stud[i].num);
printf("Name of the student %d : %s",
      i+1,stud[i].name);
printf("Marks of the student %d : %d",
      i+1,stud[i].mark);
}
getch();
}
```

10.10 ARRAYS WITHIN STRUCTURES

- C permits the use of arrays as structure members.

Example

```
#include<stdio.h>
#include<conio.h>
struct student
{
char name[20];
int num;
int mark[3];
int total;
};
struct student stud[10];
void main()
{
int i,n,j;
clrscr();
printf("\n Enter the no. of students in a class:");
scanf("%d", &n);
printf("Enter the students detials one by one\n");
for (i=0;i<n;i++)
{
printf("Enter the reg.no of the student %d :", i+1)
scanf("%d", &stud[i].num);
printf("Enter the name of the student %d :", i+1)
scanf("%s", stud[i].name);
printf("Enter the marks of the student %d :", i+1)
stud[i].total=0;
for (j=0;j<3;j++)
 {
scanf("%d", &stud[i].mark[j]);
stud[i].total= stud[i].total+stud[i].mark[j];
 }
}
for (i=0;i<n;i++)
{
printf("\nReg.no of the student %d : %d",
      i+1,stud[i].num);
printf("\nName of the student %d : %s\n",
      i+1,stud[i].name);
for (j=0;j<3;j++)
   printf("Marks of the student %d \t: %d",
      i+1,stud[i].mark[j]);
printf("\nTotal marks  of the student %d : %d",
      i+1,stud[i].total);
}
getch();
}
```

10.11 STRUCTURES WITHIN STRUCTURES

- Structure within a structure is called nested structure.

Example

```
struct salary
{
char name[20];
char dept[30];
struct
{
int da;
int hra;
int cca;
}allowance;
} employee;
```

- The salayy structure contains a member named allowance, which itself is a structure with three members.

- The members contained in the inner structure namely da, hra and cca can be reffered as

```
employee.allowance.da;
employee.allowance.hra;
employee.allowance.cca;
```

- A inner-most member in a nested-structure can be accessed by chaining all the concerned structures variables with member using dot operator.

10.12 STRUCTURES & FUNCTIONS

There are 3 methods to transferred structure from one function to another.

1. Pass each member of structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. Passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure. Therefore the necessary for the function to return the entire structure back to the calling function.
3. Using pointers to pass the structure as an argument.

- The general format of sending a copy of a structure to the called function is

function_name(structure_variable_name);

- The called function format

data_type
function_name(struct_variable_name)
{
……
……
return(expression);
}

Note

1. The called function must be declared for its type, appropriate to the data type it is expected to return.
2. The structure variable used as the actual argument and the corresponding formal parameter in the called function must be of the same struct type.

3. The return statement is necessary only when the function is returning some data back to the calling function. The expression may be any simple variable or structure variable or an expression using simple variables
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

Example

```
#include<stdio.h>
#include<conio.h>
 /* structure is defined above all functions so it
     is global. */
struct student
 {
 char name[20];
 int roll_no;
 int marks;
 };
void print_struct(struct student stu);
void main()
 {
struct student stu = {"George", 10, 69};
print_struct(stu);
return 0;
}
void print_struct(struct student stu)
 {
 printf("Name: %s\n", stu.name);
 printf("Roll no: %d\n", stu.roll_no);
 printf("Marks: %d\n", stu.marks);
 printf("\n");
}
```

10.13 UNIONS

- A **union** is a special data type available in C 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.
- Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

```
union [union tag]
 {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

- The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
- At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

Example

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

- A variable of ***Data*** type can store an integer, a floating-point number, or a string of characters.

- It means a single variable, i.e., same memory location, can be used to store multiple types of data.
- We can use any built-in or user defined data types inside a union based on your requirement.
- The memory occupied by a union will be large enough to hold the largest member of the union.

Example

- ***Data*** type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

Accessing Union Members

- To access any member of a union, we use the **member access operator (.)**.
- The member access operator is coded as a period between the union variable name and the union member that we wish to access.
- The keyword **union** to define variables of union type.

```
#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;
}
```

Output: data.str : C Programming

10.14 BIT FIELDS

- A bit field is a set of adjacent bits whose size can be from 1 to 16 bits length.
- A word can therefore be divided into a number of bit fields.
- The name and size of bit fields are defined using structure.

Syntax

struct tag-name

{

data-type name:bit-length;

data-type name:bit-length;

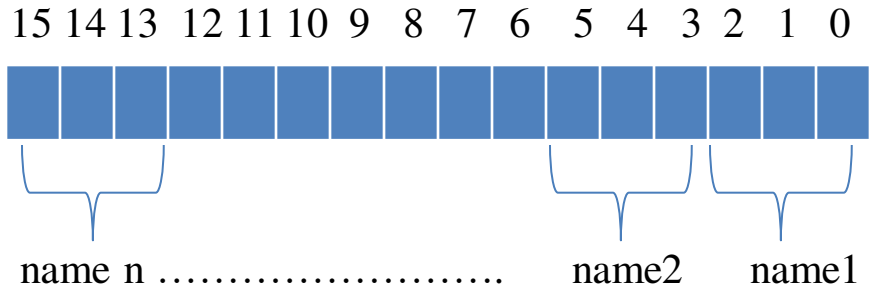
……..

……..

data-type name:bit-length;

};

- The data type is either signed int or unsigned int and the bit-length is the no. of bits used for a specified name.
- A signed bit field should have at least 2 bits(one bit for sign).
- The field name is followed by a colon.
- The bit-length is decided by the range of value to be stored.
- The largest value that can be stored is $2^{n-,}$ where n is bit-length.
- The internal representation of bit fields is machine dependent.
- It depends on the size of int and ordering of bits.



Note

1. The first field always starts with the first bit of the word.
2. A bit field cannot overlap integer boundaries. Ie. The sum of lengths of all the fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.
3. There can be unnamed fields declared with size.
4. There can be unused bits in the word.
5. We cannot take the address of a bit field variable. This means we cannot use scanf() function to read values into bit fields. We can neither use pointer to access the bit fields.
6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size.. If we try to assign larger values, behaviour would be unpredicted.

Example

```
#include <stdio.h>
#include <string.h>
struct
{
unsigned int age : 3;
} Age;
int main( )
{
Age.age = 4;
printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
printf( "Age.age : %d\n", Age.age );
Age.age = 7;
printf( "Age.age : %d\n", Age.age );
Age.age = 8;
printf( "Age.age : %d\n", Age.age );
return 0;
}
```

References

1. E. Balagurusamy, ”Programming in ANSI C”, Seventh Edition, McGraw Hill Education India Private Ltd, 2017.
2. https://www.tutorialspoint.com/cprogramming/
3. https://overiq.com/c-programming-101/structures-and-functions-in-c/