Unit 1 User-Defined Functions

What is function?

A function is a set of program statements that can be processed independently.

Function components/Function elements

Every function has the following components/elements

- Function declaration or prototype
- Function parameters
- Function definition
- Function call
- return statement

Need of Function

The **Need** of the function are

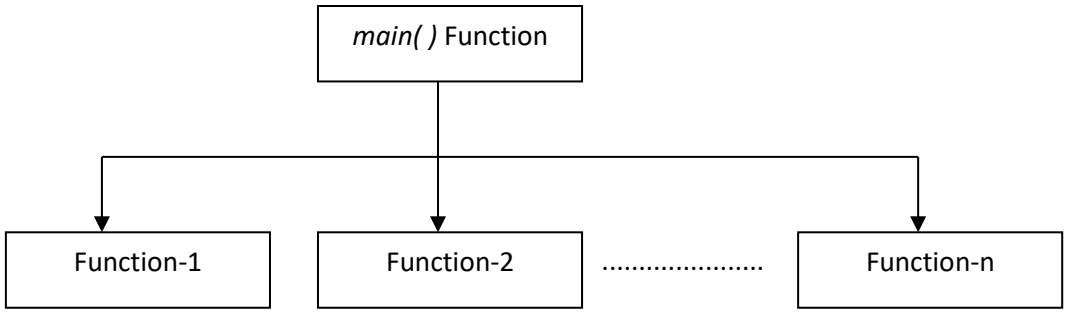
- Modular programming
- Reduction in the amount of work and development time
- Program and function debugging is easier.
- Division of work is simplified due to the use of divide-and-conquer principle.
- Reduction in size of the program due to code reusability
- Function can be access repeatedly without redevelopment, which in turn promotes reuse of code.
- Library of function can be implemented by combining well designed, tested and proven function.

User Defined Function

User-defined functions are those functions written by the user and user by him or by other user in his program. *main( )* is user-defined function. User can give any name to his user-defined function.

User Defined Function has following advantages:

1. When user writes all the code for his program into one function (*main( )* in C program) it is difficult to write, understand, debug, and test that program. So whenever we are dividing the one function into multiple parts then it is easy for user to write, understand, debug, and test the program.
2. We can use function written for one program into another program so it provides the reusability and save time and space both.
3. It provides the top-down modular programming style.



BASIC DEFINITION (TERM) USED IN C FUNCTION

In the C function we are using mainly following basic terminology:

Function Name

Each and every function has given one name and we have to follow same rules and regulation for function name whichever apply to other qualifier.

Function call

The statement which starts the execution of function is known as function call statement. We have to place this statement into another function. For example we are using one function *sum* then statement to start the execution of this function is placed in *main* function. The execution of *main* function is started by the operating system function.

Calling Function

A function which starts the execution of other function is known as calling function.

Called Function

A function which execution is started by some other function is known as the called function.

Argument (Parameter)

Argument is a value or message passed from calling function to called function. In C program we can pass any number of arguments from calling to called function. There is also possibility of function without argument.

Return value

It is a value returned by the called function to calling function. In C program return value is either 0 or 1. We cannot have more than one return value.

Components of User Defined Function

In C programming language, function has mainly three parts.

Function Declaration (Function Prototyping)

Function declaration statement is written any of the other part of the function. It tells to compiler what is function name, how many arguments we are using in function, what is data type of the argument, and what is return value from the function.

returntype functionname(datatypeofargument1, datatypeofargument2, ...);

For example, suppose we have one function to make the sum of the two numbers in which we have to pass two arguments that is *number_1* and *number_2.* Each of this value is float type. This function returns the answer that is also of type float. The name of function is *sum.*

```
float   sum(float, float);
```

In above case we can also use the argument name along with the argument data type. This argument is known as *Dummy Argument.*

```
float   sum(float number_1, float number_2);
```

Function Definition

This part of function contains executable statements within curly bracket along with argument list and return value.

```
returntype functionanme(argument1, argument2, ...., argument-n)
{
        ..................
        Block of code (Executable statements)
        ..................
}
```

For example,

```
float  sum(float number_1, float number_2)
{
        float answer;
        answer = number_1 + number_2;
        return(answer); // Return statement
}
```

To return the value from the function we have to use return statement in your function. After executing return statement the execution of function is completed and control returns to the calling function. The argument name used in function definition is known as the ***Formal Argument***.

Function call

Function call contains the statement to start the execution of function in calling function. It includes the values of the argument and assigns the value returned by the called function to some variable or at any other place.

```
variablename = functionname(valueofargument1, valueofargument2, ...);
```

Here the return value is assigned to the *variablename.* The argument used in function call statement is known as the ***Actual Argument****.*

```
void main( )
{
        float result;
        ..................
        result = sum(10.5, 20);
        ..................
}
```

CATEGORY OF FUNCTION

Based on the argument and return value we can categories function into following category:

1. Function with no argument and no return value.
2. Function with argument and no return value.
3. Function with argument and with return value.
4. Function with no argument and with return value.
5. Function that return multiple value.

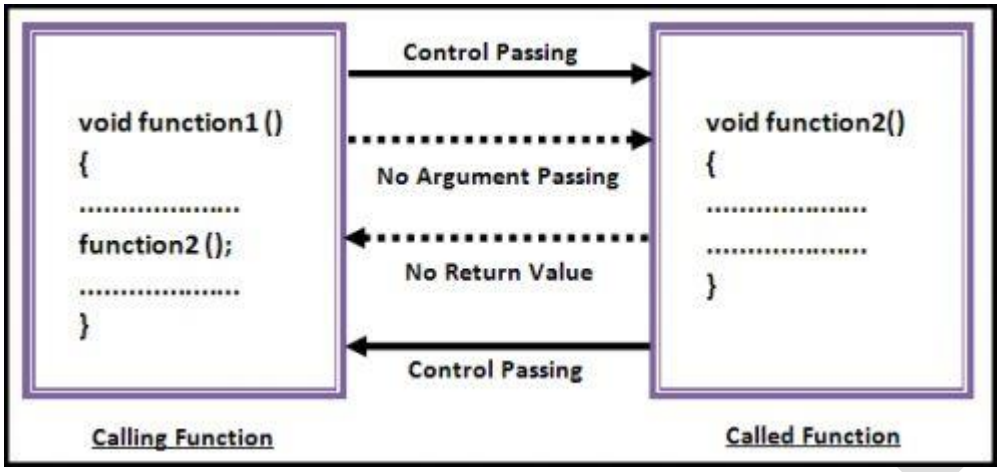
Function with no argument and no return value.

A C function without any arguments means you cannot pass data (values like int, char etc) to the called function. Similarly, function with no return type does not pass back data to the calling function. It is one of the simplest types of function in C. This type of function which does not return any value cannot be used in an expression it can be used only as independent statement. Let's have an example to illustrate this.

```
#include<stdio.h>
#include<conio.h>

void main()
{
    void printline(void);
    clrscr();
    printf("Welcome to function in C");
    printline();
    printf("Function easy to learn.");
    printline();
    getch();
}

void printline()
{
    int i;
    printf("\n");
    for(i=0;i<30;i++)
    {
            printf("-");
    }
    printf("\n");
}
```



Function with argument and no return value.

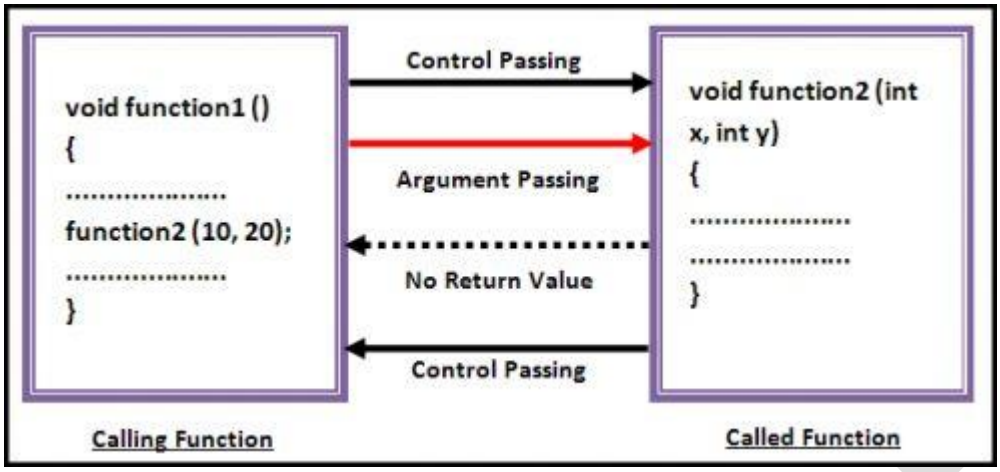
In our previous example what we have noticed that "main()" function has no control over the UDF "printfline()", it cannot control its output. Whenever "main()" calls "printline()", it simply prints line every time so the results remain the same.

A C function with arguments can perform much better than previous function type. This type of function can accept data from calling function. In other words, you send data to the called function from calling function but you cannot send result data back to the calling function. Rather, it displays the result on the terminal. But we can control the output of function by providing various values as arguments. Let's have an example to get it better.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        void add(int,int);
        clrscr();
        add(30,15);
        add(63,49);
        add(952,321);
        getch();
}

void add(int x, int y)
{
        int result;
        result = x+y;
        printf("Sum of %d and %d is %d.\n\n",x,y,result);
}
```



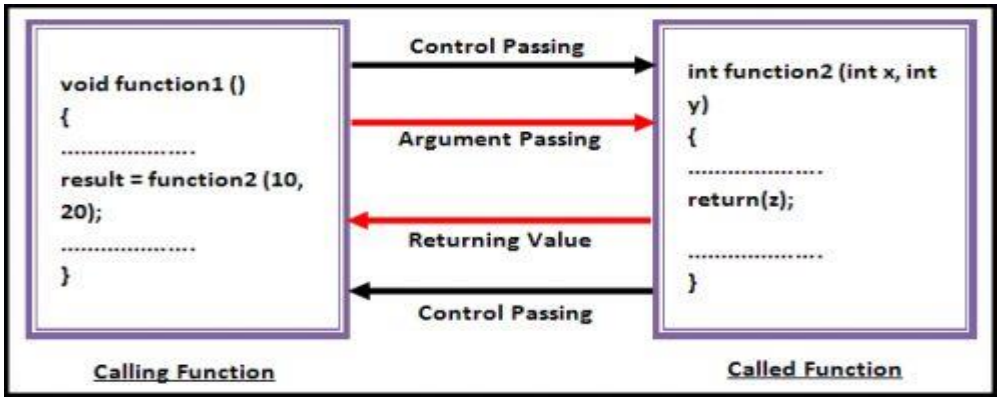
Function with argument and with return value.

This type of function can send arguments (data) from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function. And this type of function is mostly used in programming world because it can do two way communications; it can accept data as arguments as well as can send back data as return value. The data returned by the function can be used later in our program for further calculations.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        int z;
        int add(int,int);
        clrscr();
        z = add(952,321);
        printf("Result %d.\n\n",add(30,55));
        printf("Result %d.\n\n",z);
        getch();
}

int add(int x, int y)
{
        int result;
        result = x+y;
        return(result);
}
```



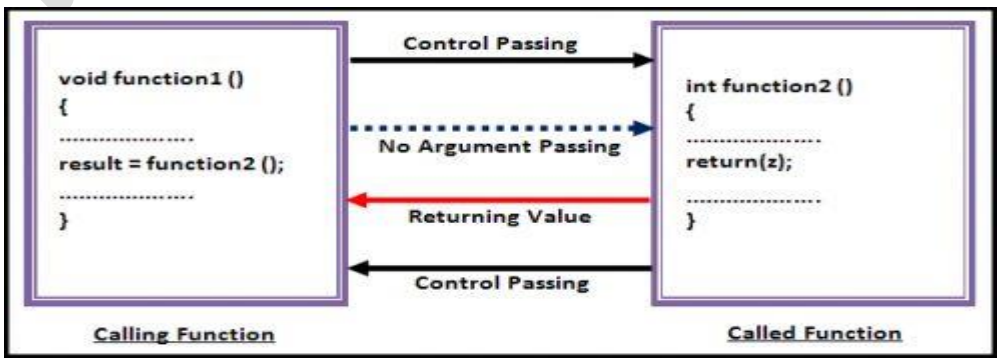
Function with no argument and with return value.

We may need a function which does not take any argument but only returns values to the calling function then this type of function is useful. The best example of this type of function is "getchar()" library function which is declared in the header file "stdio.h". We can declare a similar library function of own. Take a look.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        int z;
        int send(void);
        clrscr();
        z = send();
        printf("\nYou entered : %d.", z);
        getch();
}

int send()
{
        int no1;
        printf("Enter a no : ");
        scanf("%d",&no1);
        return(no1);
}
```



Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion()
{
  recursion(); /* function calls itself */
}
int main()
{
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Factorial

The following example calculates the factorial of a given number using a recursive function.

```
int factorial (int i)
{
        if (i <= 1)
        {
              return 1;
        }
        return i * factorial(i - 1);
}

int  main()
{
  int i = 5;
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

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

```
Factorial of 5 is 120
```

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function.

```
int fibonacci (int i)
{
        if(i == 0)
        {
                return 0;
        }

        if(i == 1)
        {
                return 1;
        }
        return fibonacci(i-1) + fibonacci(i-2);
}

int  main()
{
        int i;

        for (i = 0; i < 10; i++)
        {
                printf("%d\t", fibonacci(i));
        }
        return 0;
}
```

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

```
0       1       1       2       3       5       8       13      21      34
```

Compiled By: Mr. Navtej Bhatt, Lecturer,

BCA Department, V.P. & R.P.T.P. Science College, VV Nagar

Class: BCA SEM II

Subject: US02CBCA21 – Advanced C Programming

Declaration: This material is developed only for the reference for lectures in the classrooms. Students are required library reading for more study. This study material compiled from Book "Programming in Ansi C", by E.balaguruswamy, Third Edition., Tata McGrawHill, Publication.

Unit 2 – Structures and Unions

Basic of Structures

C supports a constructed data type known as structures, a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a 'record' in many other languages. More examples of such structures are:

Time	:	seconds, minutes, hours
Date	:	day, month, year
Book	:	author, title, price, year
City	:	name, country, population

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

Defining a Structure

Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

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

The keyword struct declares a structure to hold the details of four data fields, namely **title**, **auther**, **pages** and **price**. These fields are called structure elements or members. Each member may belong to a different type of data. **book_bank** is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.

The general format of a structure definition is as follows:

```
struct tag_name
{
        data_type  member1;
        data_type  member2;
        ....    .......    ..........
};
```

In defining a structure you may note the following syntax:

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 such as **book_bank** can be used to declare structure variables of its type, later in the program.

Array V/s 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 a structure is a programmer-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.

Declaring Structure Variables

After defining a structure format we can declare variables of that type. 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 variables names separated by commas
4. A terminating semicolon

For example, the statement

struct book_bank, book1, book2, book3;

declares **book1**,**book2** and **book3** as variables of type **struct book_bank.**

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

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

Accessing Structure Members

We can access and assign values to the members of a structure in a number of ways. The members of structure should be linked to the structure variables in order to make them meaningful members. For example, the word **title** has no meaning whereas the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the member operator '.' Which is also known as 'dot operator' or 'period operator'? For example,

```
book1.price
```

is the variable representing the price of book1 and can be treated like any other ordinary variable? Here is how we would assign values to the members of book1:

```
strcpy(book1.title,"BASIC");
strcpy(book1.author, "Balaguruswamy");
book1.pages = 250;
book1.price = 120.50;
```

We can also use scanf to give the values through the keyboard.

```
scanf("%s\n",book1.title);
scanf("%d\n",&book1.pages);
```

are valid input statements.

Structure Initialization

Like any other data type, a structure variable can be initialized at compile time.

```
struct st_record
{
    int weight;
    float height;
} student1 = {6-,180.75};

void main()
{
    struct st_record student2 = {53, 170.60};
}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variable.

Note that the compile-time initialization of a structure variable must have the following elements:

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

Rules for initializing structures

There are a few rules to keep in mind while initializing structure variables at compile-time.

1. We cannot initialize 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 as follows:
    a. Zero for integer and floating point numbers
    b. '\0' for character and strings

Arrays of Structures

We use structures to be describing the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. for example,

```
struct class student[100];
```

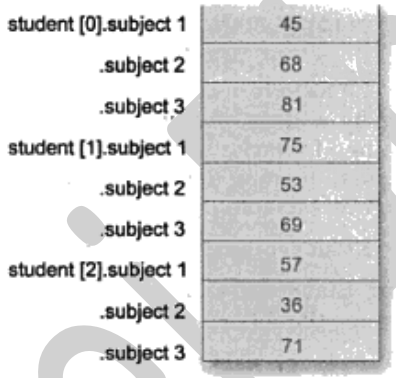
defines an array called student, that consists of 100 elements. Each element is defined to be of the type struct class. Consider the following declaration:

```
struct marks
{
        int subject1;
         int subject2;
        int subject3;
};
void main()
{
        struct marks student[3]={{45,68,81},{75,53,69},{57,36,71}};
```

This declares the student as an array of three elements student[0], student[1] and student[2] and initializes their members as follows:

```
student[0].subject1=45;
student[0].subject2=65;
        ……..
        ……..
student[2].subject3=71;
```

note that the array is declared just as it would have been with any other array. Since student is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members. Remember, each element of student array is a structure variable with three members.
Any array of structures is stored inside the memory in the same way as a multi-dimensional array. The array student actually looks as show in below figure.



Arrays within Structures

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single-or multi-dimensional arrays of type int or float. For example, the following structure declaration is valid:

```
struct marks
{
        int number;
        float subject[3];
} student[2];
```

Here, the member subject contains three elements, subject[0], subject[1] and subject[2]. These elements can be accessed using appropriate subscripts. For example, the name

```
student[1].subject[2];
```

would refer to the marks obtained in the third subject by the second student.

Structures within structures (Nested Structures)

Structure within structure means nesting of structure. We can define the new structure inside one structure or we can declare the variable of one structure inside another structure. For example in following example we have defined two structures for Address and Student.

```
struct address
{
        char society[100];
        char area[100];
        char city[50];
        int pincode;
};
struct student
{
        int rollno;
        char name[50];
        struct address add;          // Structure inside
structure
};
```

In above example we have defined one structure having member variable society, area, city, and pincode number. Then we have defined another structure having member variable roll number, name, and structure variable of structure address.

We can also define above structure by following way. Any method is valid for C program.

```
struct student
{
        int rollno;
        char name[50];

        // Structure address defined inside the structure
        struct address
        {
                char society[100];
                char area[100];
                char city[50];
                int pincode;
        } add;
};
```

We can access the members of nested structure in program as under:

```
void main( )
{
        struct student s;
        printf("Enter student detail\n");
        scanf("%d", &s.rollno);
        gets(s.name);

        // Accessing the member of nested structure
        gets(s.add.society);
        gets(s.add.area);
        gets(s.add.city);
        scanf("%d",&s.add.pincode);
}
```

Structures and Functions

We can pass the structure to the user defined function with many ways. The one method involves 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. It is, therefore, necessary for the function to return the entire structure back to the calling function.

PASSING STRCUTRE VARIABLE INSIDE FUNCTION

As like as simple variable and array variable we can also pass the structure variable inside the function. Following example demonstrate passing structure variable inside the function.

```
struct student
{
        int rollno;
        char name[100];
};
// Function declaration
void print(struct student);
void main( )
{
        struct student s;
        printf("Enter Roll No. and Name of student : ");
        scanf("%d",&s.rollno);
        gets(s.name);
        // Call a function
        print(s);
}
        // Function definition
        void print(struct student s) {
        print("Roll No. = %d\n",s.rollno);
        printf("Name = %s\n",s.name);
}
```

In above example we have defined one structure template for the student, which contain roll number and name as structure member. We have used one function print() to print the member of structure and we have passed that variable as argument inside function.

RETURNING STRUCTURE VARIABLE FROM FUNCTION

As like passing the structure variable inside structure we can return structure variable from function. But it is supported only when our compiler supports the assignment operation over structure.

```
struct student
{
int rollno;
char name[100];
};

// Function declaration
struct student read( );

void main( )
{
        struct student s;
        // Call function
        s = read( );
}

// Function definition
struct student read( )
{
        struct student t;
        printf("Enter Roll No. and Name of student\n");
        scanf("%d",&t.rollno);
        gets(t.name);
        return( t );  // Returning structure variable
}
```

In above example we have defined one structure template for the student, which contain roll number and name as structure member. We have used one function read() to read the data of structure. This function return the value of type struct student and then it is assigned to variable used inside main( ) function.

Important Points

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as struct with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument 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 ro an expression using simple variable.
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.

Unions

Unions are a concept borrowed from structure and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Union can be declared using the keyword union as follows:

```
union item
{
        int m;
        float x;
        char c;
} code;
```

This declares variable code of type union item. The union contains three members, each with a different data type. However, we can use only of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

The complier allocates a piece of storage that is large enough to hold the largest variable type in the union. To access a union member, we can use the same syntax that we use of structure members. That is,

```
code.m
code.x
code.c
```

All are valid members variables. During accessing, we should make sure that we are accessing the member whose value is currently store.

```
code.m = 379;
code.x = 7859.36;
prinft("%d",code.m);
```

Would produce erroneous output(which is machine independent).

A union creates a storage location that can be used by any one of its members at a time. When different member is assigned a new value, the new value supersedes the previous member a value.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

Compiled By: Mr. Navtej Bhatt, Lecturer,

BCA Department, V.P. & R.P.T.P. Science College, VV Nagar

Class: BCA SEM II

Subject: US02CBCA21 – Advanced C Programming

Declaration: This material is developed only for the reference for lectures in the classrooms. Students are required library reading for more study. This study material compiled from Book "Programming in Ansi C", by E.balaguruswamy, Third Edition., Tata McGrawHill, Publication.

Unit 3 – Usage of Pointer

Introduction and Usage of pointer

A pointer is a derived data type in C. it is built from one of the fundamental data types available in C. pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include:

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Addressing the address of a variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. The address can be determine with the help of the operator & available in C. The operator & immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
P = &a;
```

Would assign the address 62264 (i.e. the location of a) to the variable p. The **&** operator can be remembered as 'address of'. The & operator can be used only with a simple variable or an array.

Declaring Pointer Variables

The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

This tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. Pt_name needs a memory location.
3. Pt_name points to a variable of the data_type.

For example,

```
int *p;
```

declares the variable p as a pointer variable that points to an integer data type. Remember that the type interferes to the data type of the variable being pointed to by p and not the type of the value of the pointer. Similarly, the statement

```
float *x;
```

Declares x as a pointer to a floating-point variable.

Initialization of pointer variable

"The process of assigning the address of a variable to a pointer variable is known as initialization". All uninitialized pointers will have some unknown values that will have some unknown values that will be interrupted as memory address. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous result. It is therefore important to initialize pointer variable carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable.

For Example

```
int quantity;
int*p;                    /* declaration  */
p = &quantity;                /* initialization */
```

We can also combine the initialization with the declaration. That is,

```
int  *p =  &quantity;
```

is allowed. The only requirement here is that the variable quantity must be declared before the initialization take place. Remember, this is as initialization of pand not *p.

We must ensure that the pointer variables always point to the corresponding type of data.For example,

```
float a,b;
int x, *p;
p= &a;
b = *p;
```

will result in erroneous output because we are trying to assign the address of a **float**variable to an integer pointer. When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, since the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

```
int x, *p = &x;                    /* three in one*/
```

Is perfectly valid. It declares **x**as an integer variable and **p**as a pointer variable and then initializes **p**to the address of **x**. and also remember that the target variable **x**isdeclared first. The statement **p** to the address of **x.**
And also remember that the target variable **x** is declared first. The statement

```
int *p = &x, x;
```

is not valid.

We could also define a pointer variable with an initial value of NULL or zero. That is, the following statements are valued:

```
int *p =null;
int *p = 0;
```

With the exception of null and zero, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360; /*absolute addresses */
```

Accessing a variable through its pointer

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer. This is done by using another unary operator * (asterisk), usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p=&quantity;
n=*p;
```

the first line declares quantity and n as integer variables and p as a pointer variable pointing to an integer. The second line assigns the value 179 to quantity and the third line assigns the address of quantity to the pointer variable p. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression, the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns the value of the variable quantity, because the p is the address of quantity. The * can be remembered as ‘value at address’. Thus the value of n would 179. The two statements

```
p=&quantity;
n=*p;
```

are equivalent to

```
n=*&quantity;
```

which in turn is equivalent

```
n=quantity;
```

Pointer Expression

Like other variables. Pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then the following statements are valid.

```
Y = *p1 * *p2  same as (*p1) * (*p2)
Sum = sum + *p1;
Z = 5* - *p2/*p1;              same as (5 *(-(*p2))) / (*p1)
*p2  = *p2  + 10;
```

Note that there is a blank space between / and * in the item 3 above. The following is wrong.

```
Z = 5* - *p2/ *p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.C allows us to add integers to or subtract from pointers, as well as to subtract one pointer from another. P1+4,P2-2 and p1-p2 are all allowed. If p1 and p2 are both pointers to the same array, then **p2-p1** gives the number of element between **p1** and **p2.**

We may also use short –hand operators with the pointers.

```
P1++;
-p2;
Sum + = *p2;
```

In addition to arithmetic operations discussed above, pointers can also be compared using the relational operators. The expression such as **p1>p2**, **p1==p2** and **p1!=p2**are allowed. However, any comparison of pointers
That refer to separate and unrelated variables makes no sense. Comparison can be used meaningfully in handling arrays and strings.We may not use pointers in division or multiplication. For example , expression such as

```
p1/p2or p1 * p2 or p1/3
```

are not allowed. Similarly, two pointers cannot be added. That is, **p1+p2**is illegal.

Pointer increments and scale factor

We have seen that the pointer can be incremented like

```
p1 = p2 + 2;
p1 = p1 + 1;
```

And so on. Remember, however an expression likep1++;

Will cause the pointer **p1** to point to the nest value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1+1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the *scale factor.*

Rules for Pointer Operations

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the value of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subscribed from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e&x=10; is illegal).

Pointers and Arrays

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array.

The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array x as follows:

```
int x[5] = {1,2,3,4,5};
```

suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will stored as follows:

Element	Value	Address
X[0]	1	1000
X[1]	2	1002
X[2]	3	1004
X[3]	4	1006
X[4]	5	1008

The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored. That is

X =&x[0] = 1000

If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the following assignment:

p = x;

This is equivalent to

p =&x[0];

now, we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown as:

p = &x[0] (=1000)
p+1 =&x[1] (=1002)
p+2 =&x[2] (=1004)
p+3 =&x[3] (=1006)
p+4 =&x[4] (=1008)

you may notice that the address of an element is calculated using its index and the scale factor of the data type.

Pointers and Structures

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose product represents the address of its zeroth element. Consider the following declaration :

```
struct inventory
{
        char name[30];
        int number;
        float price;
 } product[2], *ptr;
```

This statement declares product as an array of two elements, each of the type struct inventory and ptr as a pointer to data objects of the type inventory. The assignment,

ptr = product;

Would assign the address of the zeroth element of product to ptr. That is, the pointer ptr will now point to product[0]. Its members can be accessed using the following notation.

ptr → name
ptr → number
ptr → price

The symbol → is called the arrow operator is made up of a sign and a greater than sign. Note that ptr → is simply another way of writing product[0]. When the pointer ptr is incremented by one, it is made to point to the next record. i.e. product[1].

The following for statement will print the values of members of all the elements of product array.

```
for(ptr=product; ptr<product+2;ptr++)
{
        printf("%s %d %f\n",ptr->name,ptr->number,ptr->rate);
}
```

Pointers as function arguments

When an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If x is an array, when we call any function by passing x as an argument like sort(x), the address of x[0] is passed to the function sort. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal way.

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as 'call by reference', where the process of passing the actual value of variable is known as 'call by value'. The function which is called by 'reference' can change the value of the variable used in the call.

Consider the following code:

```
void main()
{
        int x;
        x = 20;
        change(&x);         /* call by reference or address */
        printf("%d\n",x);
}
change(int *p)
{
        *p=*p+10;
}
```

When the function change() is called, the address of the variable, not its value, is passed into the function change(). Inside change(), the variable p is declared as a pointer and therefore p is the address of the variable x. The statement,

```
*p = *p + 10;
```

means'add 10 to the value stored at the address p'. since p represents the address of x, the value of x is changed from 20 to 30. Therefore the output of the program will be 30, not 20.

This mechanism is known as "**call by address**" or "**pass by pointers**".

Rules for pass by Pointers

1. The types of the actual argument and formal arguments must be same.
2. The actual arguments (in the function call) must be the address of variables that are local to the calling function.
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 *.

Returning multiple values through pointer

In functions, we have seen that it return just one value using a return statement. That is because; a return statement can return only value. Suppose, however that we want to get more information from a function. We can achieve this 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**.

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

```
void main()
{
        int x=20,y=10,s,d;
        void mathoperation(int x, int y, int *s, int *d);
        clrscr();
        mathoperation(x,y,&s,&d);
        printf("\n s=%d \n d=%d",s,d);
        getch();
}
void mathoperation(int x, int y, int *s, int *d)
{
        *s=x+y;
        *d=x-y;
}
```

Dynamic memory allocation

In programming we may come across situations where we may have to deal with data, which is dynamic in nature. The number of data items may change during the executions of a program. The number of customers in a queue can increase or decrease during the process at any time. When the list grows we need to allocate more memory space to accommodate additional data items. Such situations can be handled move easily by using dynamic techniques. Dynamic data items at run time, thus optimizing file usage of storage space.

The process of allocating memory at run time is known as dynamic memory allocation. Although c does not inherently have this facility there are four library routines which allow this function.

Many languages permit a programmer to specify an array size at run time. Such languages have the ability to calculate and assign during executions, the memory space required by the variables in the program. But c inherently does not have this facility but supports with memory management functions, which can be used to allocate and free memory during the program execution. The following functions are used in c for purpose of memory management.

Function	Task
malloc	Allocates memory requests size of bytes and returns a pointer to the 1st byte of allocated space
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space.

Memory Allocation Process

According to the conceptual view the program instructions and global and static variable in a permanent storage area and local area variables are stored in stacks. The memory space that is located between these two regions in available for dynamic allocation during the execution of the program. The free memory region is called the heap. The size of heap keeps changing when program is executed due to creation and death of variables that are local for functions and blocks. Therefore it is possible to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions mentioned above will return a null pointer.

Allocating a block of memory: malloc

A block mf memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr=(cast-type*)malloc(byte-size);
```

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

Example:

```
x=(int*)malloc(100*sizeof(int));
```

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int.The malloc allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a NULL.

Allocating multiple blocks of memory: calloc

Calloc is another memory allocation function that is normally used for requesting memory space at runtime for storing derived data types such as arrays and structures. While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size and then sets all bytes to zero. The general form of calloc is:

```
ptr=(cast-type*) calloc(n,elem-size);
```

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

The following segment of a program allocates space for a structure variable:

```
.....
structstudent
{
        char name[25];
        float age;
        longintid_num;
};
typedefstruct student record;
record *st-ptr;
intclass_size = 30;

st_ptr=(record *)calloc(class_size, sizeof(record));
.....
```

record is of type struct student having three members: name, age and id_num. the calloc allocates memory to hold data for 30 such records.

Releasing the used space: free

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing

any other information, we may release that block of memory for future use, using the free function.

```
free(ptr);
```

ptr is a pointer that has been created by using malloc or calloc.

Altering the size of a block: realloc

It is likely that the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both cases, we can change the memory size already allocated with the help of the function realloc. This process is called the reallocation of memory. For example, if the original allocation is done by the statement

```
ptr = malloc(size);
```

then reallocation of space may be done by the statement

```
ptr = realloc(ptr, newsize);
```

this function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block. The newsize may be larger or smaller than the size.

Compiled By: Mr. Navtej Bhatt, Lecturer,

BCA Department, V.P. & R.P.T.P. Science College, VV Nagar

Class: BCA SEM II

Subject: US02CBCA21 – Advanced C Programming

Declaration: This material is developed only for the reference for lectures in the classrooms. Students are required library reading for more study. This study material compiled from Book "Programming in Ansi C", by E.balaguruswamy, Third Edition., Tata McGrawHill, Publication.

Unit 4 – Usage of File Handling

Introduction

Function scanf and printf are used for read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This work fine as long as the data is small. Many real life problems involve large volume of data and in such situations; the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volume of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. The method employs the concept of files to store the data. A file is a place on the disk where a group of related data is stored. Like most other languages, c supports a number of functions that have the ability to perform basic file operations, which include

- Naming a file,
- Opening a file,
- Reading a data from a file,
- Writing a data to a file and
- Closing a file.

There are two distinct ways to perform the file operation in c. the first one is known as the low level I/O and uses UNIX system calls. The second method is referred to as the high-level I/O operation and uses function in C's standard I/O library. There are listed in table.

Function Name	***Operation***
fopen()	Creates a new file for use. Opens an existing file for use.
fclose()	Closes a file which has been opened for use.
getc()	Reads a character from a file.
putc()	Writes a character to a file.
fprintf()	Writes a set of data values to a file.
fscanf()	Reads a set of data values from a file.
getw()	Reads an integer from a file.
putw()	Writes an integer from a file.

There are many other functions. Not all of them are supported by all compilers.

Defining and Opening a File

If we want to store data in a file in the secondary memory, we must specify certain things about the file to the operating system. They include:

1. File Name.
2. Data Structure
3. Purpose

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension.

Examples:

Input.data
Store
Prog.c
Student.c
Text.out

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. **FILE** is a defined data type. When we open a file, we must specify what we want to do with the file. For Example, we may write data to the file or read the already existing data. Following is the general format for declaring and opening a file:

```
FILE *fp;
fp = fopen("filename", "mode");
```

The first statement declared the variable fp as a "pointer to the data type **FILE**" is a defined data type. File is structured that is defined in I/O library. The second statement opens the file named filename and assigns an identifier to the **FILE** type pointer fp. This pointer which contains all the information absolute the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following.

r Open the file for reading only.
w Open the file for writing only.
a Open the file for appending (or editing) data to it.

Both the filename and mode are specified as strings. They should be enclosed in double quotation marks. When trying to open a file, one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The content is detected, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Consider the following statements:

```
FILE  *p1, *p2;
p1  = fopen("data", "r");
p2  = fopen("results", "w");
```

The file **data** is opened for reading and results is opened for writing. In case, the **result** file already exists, its content is detected and the file is opened as anew file. If **data** file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

r+ the existing a file is opened to the beginning for reading and writing.
w+ same as w except both for reading and writing.
a+ same as a except both for reading and writing.

Closing a File

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose (file_pointer);
```

This would close the file associated with the **FILE** pointer file_pointer. Look at the following segment of a program.

```
-----
FILE   *p1, *p2;
p1 = fopen ("INPUT", "w");
p2 = fopen ("OUTPUT", "r");
-----
fclose (p1);
fclose (p2);
```

This program opens two files and closes them after all operation on them are completed. Once a file is closed, its file pointer can be reused for another file. As a matter of fact all files are closed automatically whenever a program terminates.

INPUT/OUTPUT Operations on Files

Once a file is opened, reading out of writing to it is accomplished using the statement I/O routines.

The getc and putc functions

The simplest file I/O functions are getc and **putc.** These are analogous to **getchar** and **putchar functions** and handle one character at a time. Assume that a file is opened with mode w and file pointer **fp1.** Then, the statement

```
putc(c, fp1);
```

writes the character combined in the character variable **c** to the file associated with **FILE** pointer fp1. Similarly, getc is used to read a character from a file that has been opened in read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of **getc** and **putc.** The **getc** will return an end-of-file maker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

The getw and putw functions

The **getw** and **putw** are integer-oriented functions. They are similar to the **getc** and **putc** functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of **getw** and **putw** are:

```
putw(integer,fp);
getw(fp);
```

The fprintf and fscanf functions

Most compilers support two other functions, namely fprintf and fscanf, that can handle a group of mixed data simultaneously.

The functions fprintf and fscanf perform I/O operations that are identical to the familiar printf and scanf functions, expect of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of fprintf is

```
fprintf(fp,"control string",list);
```

where fp is a file pointer associated with a file that has been opened for writing. The control string contains output specifications for the items in the list. The list may include variables, constants and string.

Example:

```
fprintf(f1,"%s %d %f",name,age,7.5);
```

here, name is an array variable of type char and age is an int variable.

The general format of fscanf is

```
fscanf(fp,"control string",list);
```

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the control string.

Example:

```
fscanf(f2,"%s %d", item, &quantity);
```

like scanf, fscanf also returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

Error Handling during I/O Operations

While dealing with files,it is possible that an error may occur. This error may occur due to following reasons:

- Reading beyond the end of file mark.
- Performing operations on the file that has not still been opened.
- Writing to a file that is opened in the read mode.
- Opening a file with invalid filename.
- Device overflow.

Thus, to check the status of the pointer in the file and to detect the error is the file. C provides two status-enquiry library functions

feof() - The feof() function can be used to test for an end of file condition

Syntax

```
feof(FILE *file_pointer);
```

Example

```
if(feof(fp))
    printf("End of file");
```

ferror() - The ferror() function reports on the error state of the stream and returns true if an error has occurred.

Syntax

```
ferror(FILE *file_pointer);
```

Example

```
if(ferror(fp)!=0)
    printf("An error has occurred");
```

Compiled By: Mr. Navtej Bhatt, Lecturer,

BCA Department, V.P. & R.P.T.P. Science College, VV Nagar

Class: BCA SEM II

Subject: US02CBCA21 – Advanced C Programming

Declaration: This material is developed only for the reference for lectures in the classrooms. Students are required library reading for more study. This study material compiled from Book "Programming in Ansi C", by E.balaguruswamy, Third Edition., Tata McGrawHill, Publication.