Programming Principle & Algorithm

Class- BCA Ist Semester

Dr. Dharm Raj Singh

Assistant Professor, (HOD)

Department of Computer Application

JagatpurP. G. College, Varanasi

Mobile No. 9452070368, 7275887513

Email- dharmrajsingh67@yahoo.com

Outline

1. **MODULE 4**

- Unit 1 : Functions
- Unit 2 : Variables and Storage Classes

Functions

- In C a large program can be divided into a number of smaller, complete and independent subprograms. These subprograms possess self-contained components, each of which has some unique, identifiable purpose. This task is called **modularization** and each sub program is called a **module** or a **function.**
- The function takes a data from main () function and returns a value. To invoke a function call is made in the main () function.
- Thus a C program can be ***modularized through the*** intelligent use of such functions. This kind of approach to program development is called **modular approach**.

Procedure-Oriented Programming

- In the procedure oriented approach, the problem is viewed as the sequence of things to be done such as reading, calculating and printing such as cobol, fortran and c. The primary focus is on functions. A typical structure for procedural programming is shown in fig.

➢ **Some Characteristics exhibited by procedure-oriented programming are:**

- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

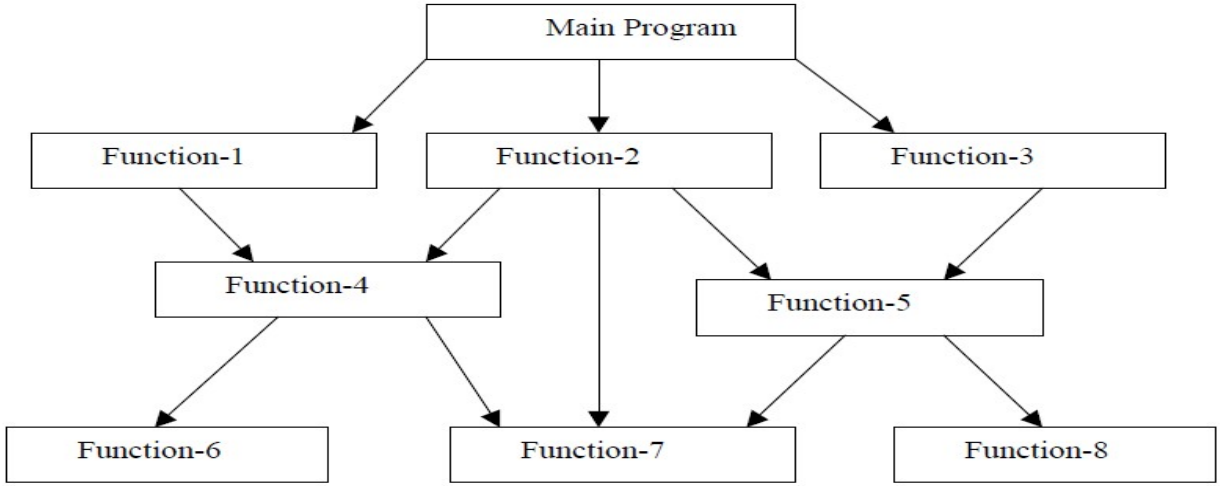


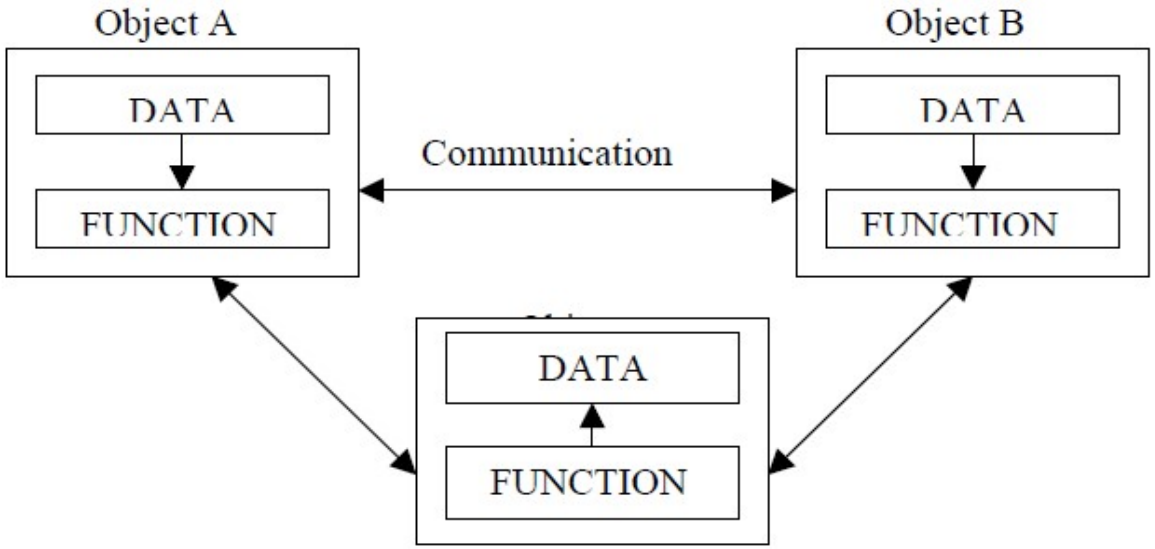
Fig. 1.2 Typical structure of procedural oriented programs

Object Oriented Paradigm

- OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig.1.3.

➢ **Some of the features of object oriented programming are:**

- Programs are divided into what are known as objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows botton



Classification

C functions can be classified into two categories, namely library functions and user-defined functions. The library functions are standard functions available within C-Language library (built in functions) but user defined function are functions that are created by the user. User defined functions (UDF) usually use library functions to get the job done. Example of library functions are printf and scanf and main is an example of user defined function. This chapter deals in detail with user defined functions.

Structure of Function

The structure and usage of functions can be well understood by dividing it into three sections namely:

1. Function declaration
2. Function invocation (calling)
3. Function definition

Calling a Function

Define the function before it is called. In this approach the function called is defined before the main () function. i.e. defining before main(). Syntax is

Function definition[function header + function body]

```
main()
{
function();
}
```

Function Definition

The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
body of the function
}
```

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return_type** is the keyword **void**.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

```
/* void function example*/
#include <istdio.h>
void  raj ()
{
Printf( “I am learning functions in c“);
}
int main ()
{
raj ();
return 0;
}
```

```
#include <stdio.h>
int max(int num1, int num2);     /* function declaration */
int main ()
{
int a = 100;
int b = 200;
int ret;
ret = max(a, b);              /* calling a function to get max value */
printf( "Max value is : %d\n", ret );
return 0;
}
int max(int num1, int num2)  /*function returning the max between two numbers*/
{
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Function Prototypes

All identifiers in C need to be declared before they are used. This is true for functions as well as variables. For functions the declaration needs to be done before the first call of the function. There are two ways to do this. They are

1. **Define the function before it is called**
2. **Declare the function before it is called.**

Function Prototypes: In this approach the function called is defined before the main () function. ie defining before main(). Function definition comprises of function header and the function body. The function definition itself can act as an implicit function declaration. Hence the complier knows what functions are used in main() before entering the main().

Define the function before it is called

```
int sum (int n1, int n2)
{
int answer;
answer = n1 + n2;
return answer;
}
main ()
{
int number1, number2, total;
printf (“input two number");
scanf ("%d %d", &number1, &number2);
total = sum (number1, number2);
printf ("The sum of %d plus %d is %d\n", number1,number2, total);
}
```

Function Prototypes

Function declaration tells the compiler," a function that looks like this is coming up later in the program", so it is like seeing reference of it before the function itself.

To **declare** a function prototype simply state the data type the function returns, the function name and in brackets list the type of parameters in the order they appear in the function definition. Function prototype contains the same information as the function header contains, but it ends with semicolon. The only difference between the header and the prototype is the semicolon (;). There must the semicolon at the end of the prototype.

```
int sum (int, int); /* Function prototype*/
int main (void)
{
int total;
total = sum (2, 3);    /* Function call*/
printf ("Total is %d\n", total);
return 0; }
int sum (int a, int b) /* Function header*/
{
return a + b;
}
```

Actual Parameters: The arguments or parameters used in the calling function to call the called function are called as the Actual parameters.

Formal parameters: The arguments or parameters used in the called function header are called as the Formal parameters. Formal arguments are the arguments available in the function definition.

```
Void add (int a, int b)     / *Formal parameters */
{
int c;
c=a+b;
printf (“The sum is %d”,c);
return;
}
main()
{
int x,y;
Printf(“ Enter two integers“)
scanf(“%d %d” ,&x, &y);
add (x, y);        / *Actual parameters */
}
```

Types of Functions

According the arguments and the returning value, functions are divided into three categories.

1) A function with no arguments and no return value

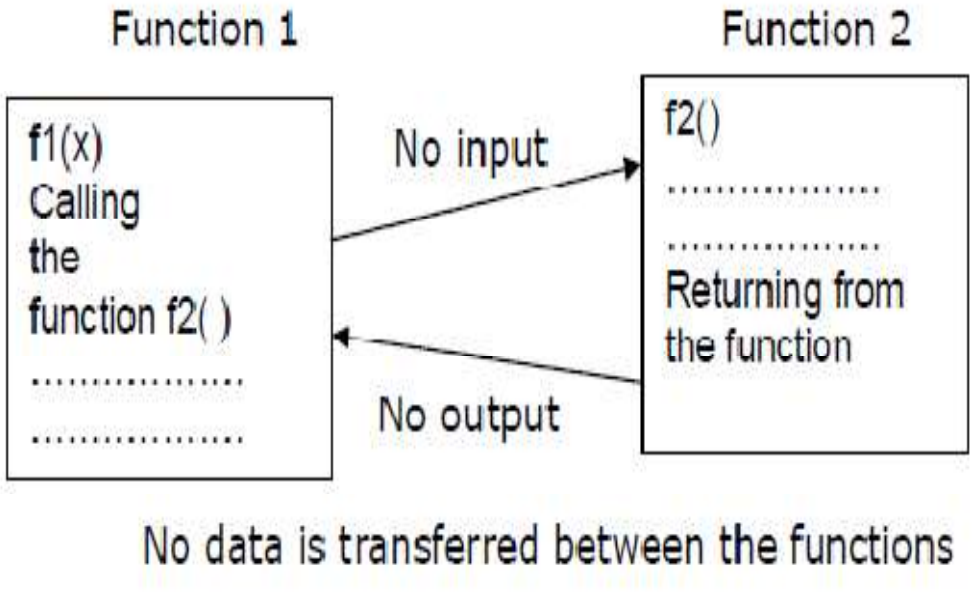
2) A function with no arguments and return a value

3) A function with an argument or arguments and returning no value

4) A function with arguments and returning a value.

A Function With No Arguments And No Return Value

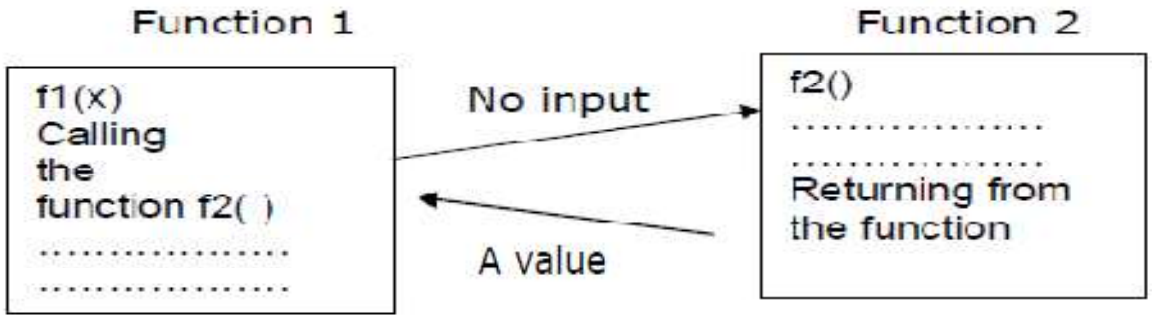
If a called function does not have any arguments, it is not able to get any value from the calling function. Also if it does not return any value, the called function is not receiving any value from the function when it is called.




```
Void message();
main( )
{
message ( );
}
void message( )
{
printf("\n WELCOME");
}
```

A Function with No Arguments and Returns a Value

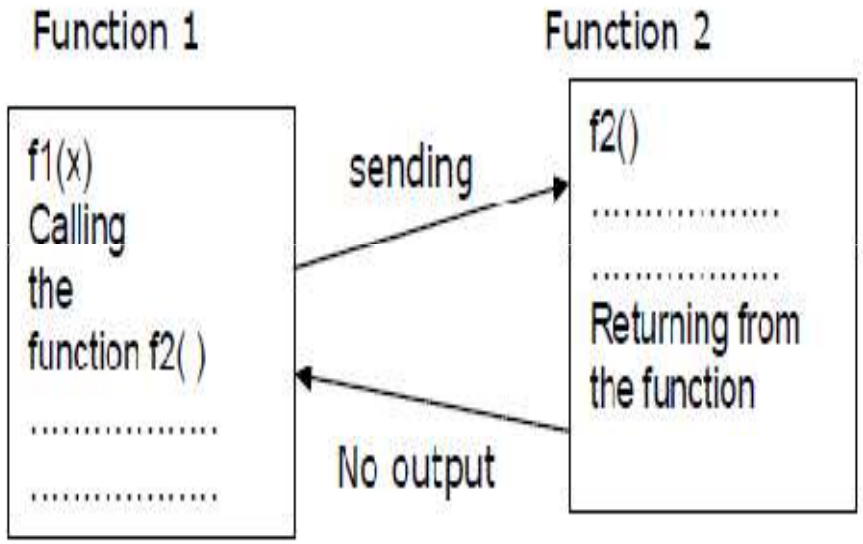
A function which does not get value from the calling function but it can return a value to calling program.




```
int temp( ) ;
main( )
    {
    int a;
    a=temp( );
    printf("\n The returned value form the function is %d", a);
    }
    int temp( )
    {
    return(100);
    }
```

A Function with Arguments and Returns No Value

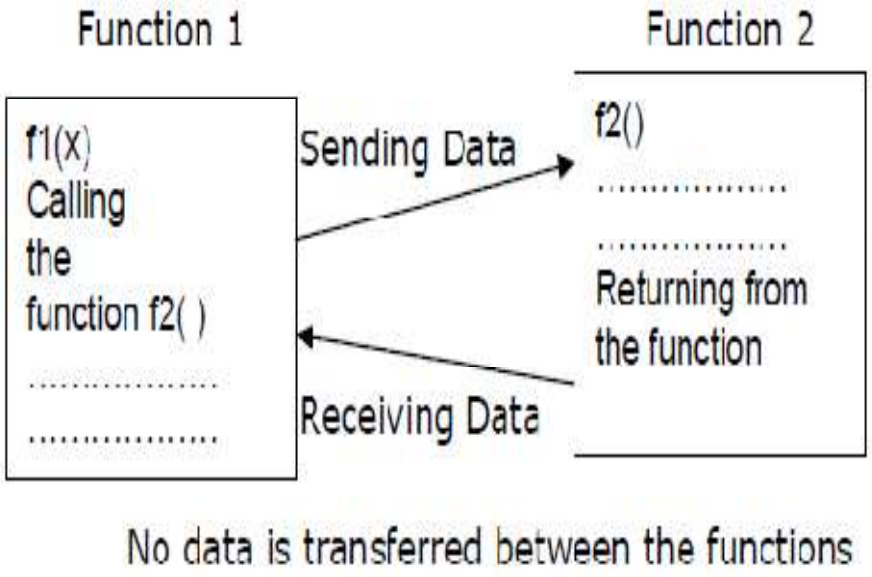
A function has an argument or set of arguments. Through arguments calling function can pas values to function called. But calling function does not receive any value.




```
Int largest(int, int);
main()
{
int a,b;
printf("Enter the two numbers");
scanf("%d%d",&a,&b);
largest(a, b);
}
largest(int a, int b)
{
if(a>b)
printf("Largest element=%d",a);
else
printf("Largest element=%d",b);
}
```

A Function with Arguments and Returning a Value

Here arguments are passed by caller function to called function. Called function returns value to caller function. Called function can receive any number of arguments but can *return only one result*.




```
float sum(float a, float b);
main()
{
float a=5,b=15,result;
result=sum(a,b);
printf("average=%f \ n",result);
}
float sum(float a, float b)
{
float s;
s=a+b;
return(s);
}
```

Function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

```
void swap(int x, int y);
int main ()
{
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
swap(a, b);     /* calling a function to swap the values */
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
void swap(int x, int y)
{
int temp;
temp = x;      /* save the value of x */
x = y;          /* put y into x */
y = temp;     /* put x into y */
return;
}
```

Function call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

```
void swap(int *x, int *y);
int main ()
{
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
swap(&a, &b);      /* calling a function to swap the values. */
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
}
void swap(int *x, int *y)
{
int temp;
temp = *x;      /* save the value at address x */
*x = *y;        /* put y into x */
*y = temp;      /* put x into y */
return;
}
```

Recursion

Recursion is the process in which a function repeatedly calls itself to perform calculations.

```
int fact(int);
 void main()
 {  int f, int n;
 printf("\nENTER A NUMBER: ");
 scanf("%d",&n);
 f=fact(n);
 printf("\nFACTORIAL OF %d IS %d", n, f);
 }
 int fact(int a)
 { int fac;
 if(a==1 || a==0)
 return(1);
 else
 fac=a*fact(a-1);
 return(fac);
 }
```

Storage Class

Storage class is a concept in c which provides information about the variable's visibility, lifetime and scope. The meaning of each term is as follows.

Scope: Scope is the region in which a variable is available for use.

Visibility: The program's ability to access a variable from memory is called as visibility of variable.

Lifetime: The lifetime of the variable is duration of the time in which a variable exists in the memory during execution.

1. Local or Automatic variables
2. Global or External variables
3. Static Variables
4. Register Variables

Local or Automatic Variables

Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits. Automatic variables are local or private to a function in which they are defined. Other names of automatic variable are *internal variable* and *local variable.* Default initial Value: Garbage.

```
main()
{
auto int x, y;
int a,b;
x = 10;
printf("Values : %d %d", x, y);
}
```

Global or External Variables

External variable is a global variable which is declared outside the function. The memory cell is created at the time of declaration statement is executed and is not destroyed when the flow comes out of the function to go to some other function. **Default initial Value: Zero**

```
int x = 100;
main ( )
{
x = 200;
f1 ( );
f2 ( );
}
f1 ( )
{
x = x + 1;
}
f2 ( )
{
x = x + 100;
Printf(" The final value of global variable x is %d",x );
}
```

Static Variables

These variables are alive throughout the program. A static variable can be initialized only once at the time of declaration. The initialization part is executed only once and retains the value till the end of the program. **Default initial Value: Zero.**

```
Void stat();
main()
{
int j;
for(j=1; j<=3; j++)
stat();
}
Void stat()
{
static int x=0;
x=x+1;
printf("The Value of X is %d\n",x);
}
```

Register Variables

A variable is usually stored in the memory but it is also possible to store a variable in the processor's register by defining it as register variable. The registers access is much faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster. Since only a few variables can be placed in a register, it is important to carefully select the variables for this purpose. However c will automatically convert register variables into normal variables once the limit is exceeded. **Default initial Value: Garbage.**

```
main ( )
{
register x , y z;
Printf(" Enter two numbers");
scanf("%d %d",&x,&y);
z=x+y;
printf("\n The sum is %d",z);
}
```

Analysis:

In the above program, all the variables are stored in the registers instead of memory.

Exercise

1. Write a function to calculate the factorial.
2. Write a function power (a, p), to calculate the value of a raised to p
3. Write a function to determine whether the year is a leap year or not.
4. Write a function to obtain the prime factors of this number.
5. Write a function to obtain the LCM and HCM of input any two number.
6. Write a program to print Fibonacci series using function.
7. Write a program to check input number prime or not using function.
8. Write a program to check input number perfect or not using function.
9. Write a program to check input number even or not using function.
10. Write a program to print factorial using recursion.
11. Write a program to print Fibonacci series using recursion.
12. Write a program to print factorial using function.
13. Write a program to find LCM of input any two number using recursion.
14. Write a program to find HCM of input any two number using recursion.

References

- Kanetkar, Yashavant P. "Let UsC Fifth Edition." (2017).
- Kernighan, Brian W., and Dennis M. Ritchie. *The C programming language*. 2006.
- Ritchie, Dennis M., Brian W. Kernighan, and Michael E. Lesk. The C programming language. Englewood Cliffs: Prentice Hall, 1988.
- McGraw-Hill, Herbert Schildt Tata. "The Complete Reference C fourth Edition”." (2005).
- Griffiths, David, and Dawn Griffiths. Head First C: A Brain-Friendly Guide. " O'Reilly Media, Inc.", 2012.
- Programming in C-Balguruswamy

Declaration

“The content is exclusively meant for academic purpose and for enhancing teaching and learning. Any other use for economic/commercial purpose is strictly prohibited. The users of the content shall not distribute, disseminate or share it with anyone else and its use is restricted to advancement of individual knowledge. The information provided in this e-content is authentic and best as per knowledge”.

Dr. Dharm Raj Singh
Assistant Professor, (HOD)
Department of Computer Application
JagatpurP. G. College, Varanasi

Thank you