Lesson3: Procedural Abstraction And Functions

Objectives

After completing this lesson you will understand about:

- The functions and types of functions.
- How to write functions.
- Local variables, void functions, function overloading, inline functions.
- Parameter passing, passing default arguments and function calling another functions.
- Recursive functions
- Procedural abstraction, testing and debugging.

Structure Of The Lesson

3.1 Functions
    3.1.1 Predefined functions
    3.1.2 Programmer defined functions
3.2 Local variables
3.3 Parameter passing into functions
        3.3.1 Call by value mechanism
        3.3.2 Call by reference mechanism
3.4 Function Overloading
3.5 Void functions
3.6 Function calling another function
3.7 Inline functions
3.8 Default arguments
3.9 Recursive functions
3.10 Procedural abstraction
3.11 Testing and debugging
3.12 Summary
3.13 Technical terms
3.14 Model questions
3.15 References

3.1 Functions

C++ has facilities to include separate subprograms into programs. These subprograms are called functions. A function that returns a value is like a small program. The arguments to the function serve as the input to this small program and the value returned is like output of this program. When a subtask of a program take some values as input and produce a single value as its result, then such subtask can be said as a function.

Functions can be divided into two. They are

a) Predefined functions
b) Programmer defined functions.

3.1.1 Predefined Functions

Predefined functions are the functions that are already built and supplied along with the compiler. These functions are also called library functions and they are stored in the header files with ".h" extension. A header file for a library provides the compiler with certain basic information about the library and **include** directive delivers this information to the compiler. Some of the header files are – iostream.h, math.h, ctype.h,stdlib.h, string.h, manip.h, conio.h some predefined math functions are:

Return type	Function name	Example	O/p
double	sqrt(double n)	sqrt(4.0)	2.0

This returns the square root of a given number

Return type	Function name	Example	O/p
double	pow(double a,double b)	pow(2.0,3.0)	8.0

This returns the a^b . If the first argument is a negative number, then the second argument must be a whole number.

Return type	Function name	Example	O/p
double	fabs(double num)	fabs(-7.5)	7.5

It calculates the absolute value of the given number

double	ceil(double num)	ciel(3.2)	4

It returns the smallest integer greater than the number.

double	floor(double num)	floor(3.9)	3

It returns the largest integer less than the number.

3.1.2 Programmer Defined Functions

A function can be defined by the user either in the same file as part of the main program or in a separate file so that the function can be used several times. Such functions are called **Programmer defined functions**. A function is like a small program is same as running the program. A function generally uses formal parameters to input the various values into the function. The description of the function is divided into two parts. They are called

- function prototype
- function definition

Function Prototype: The prototype describes what the function look like i.e., it tells about the function name the return type of the function, the number of arguments and the type of the arguments passed into the function, the identifier name may or may not be declared in the arguments list of the prototype. The identifiers used in the arguments list are called as formal parameters. The formal parameters are used as a kind if place holder for the arguments. The general format of the function prototype is:

Syntax:
Type returned function name(type1 fp1,jtype2 fp2);

e.g.: double totalweight (int num, double weight);
or
double totalweight (int, double);

A function prototype should appear before the function call, normally they are placed before the main part of the program.

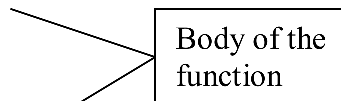
Function Definition: A function definition describes how the function computes the value it returns. A function definition consists of a function header followed by the function body. A function header is written same as the function prototype, except the header does not have a semicolon at the end. The function body follows the function header and completes the definition. The function body consists of declaration and executable statements enclosed within a pair of braces. the value returned by the function is determined when the function executes a return-statement.

The general format of a function definition is given below.

```
Return type function name (data type fpar1,datatype,fpar2…)
{
declaration
:
:
executable statements

:
:
return statement;
}
```

```
eg:  float billcal(int n, float c)   //function header
{
const float tax=0.15;
float bill amount;
billamount=n*c;
bill=billamount+billamount*Tax;
return bill;
}
```



Function Call: Function call is a statement that involves a controlled transfer to the definition of the function, i.e., invokes the function. The function call statement consists of function name and list of actual arguments written within the parameters.

Syntax: Function name (actual parameters list);

Write a program to display maximum of two numbers using functions.

```
#include<iostream.h>
void main()
{
int a,b;
int max(int,int);
cout<<"enter a,b";
cin >> a>>b;
cout<<maxi(a,b);
getch();
return 0;
}
int max(int a, int b)
{
if(a>b)
return a;
else
return b;
}
```

3.1.2 Local Variable

Variables that are declared within the body of the function are said to be local to that function or the scope is within the function. The variables that are declared in the main part of the program are said to be local to the main part of the program.

If variable is local to a function then we can have another variable with the same name which is declared in the main part of the program or in some other function. These two will be different variables even though they have the same name.

Program to demonstrate the local and global variables

```
#include<iostream.h>
#include<conio.h>
int x=5;//global variable
int fun1( );
int fun2( );
int fun3( );
```

```
void main( )
{
cout<<x<<end1;
int  x=10;          //local to main
cout<<fun1( )<<end1;
cout<<fun2( )<<end1;
cout<<fun3( )<<end1;
cout<<x;
return 0;
}
int fun1()
{
x+=2;   //uses global variable
cout<<x<<"\n";
return x;
}
int fun2( )
{ int x;  //local to fun2()
cin>>x;
cout<<x<<"\n";
return x;
}
int fun3( )
{
 x=x+1;  //uses global variable
cout<<x<<"\n";
return x;
}
```

3.3 Parameter Passing Into Functions

The parameters to a function can be passed in two ways. They are

- Call by value
- Call by reference

Call By Value Mechanism: Formal parameters of a function are local to the function. They can be used just like a local variable. We should not add a variable declaration for the formal parameters. When a function is called the values of actual arguments are copied to formal parameters within the function. We can make any changes to the values of the formal parameters but these changes will not be reflected on the values of the actual arguments. This mechanism is called call by value mechanism.

Program to demonstrate call by value mechanism

```
#include<iostream.h>
void swap (int,int);
void main()
{
int a,b;
cout<<"enter a,b";
cin >> a>>b;
swap (a,b);
}
void swap(int a, int b);
{
int c;;
c=a;
a=b;
b=c;
cout<<a<<" "<<b;
}
#include<iostream.h>
void swap (int,int);
```

output:

```
enter a,b2
3
3 2
```

Call By Reference Mechanism: To make a formal parameter a call by reference parameter an "&" sign should be appended to its date type name. The corresponding actual argument in the function call should be a variable but not a constant or an expression when the function is called the corresponding actual variable argument (not a value) will be substituted for the formal parameter. Any changes made to the formal parameter in the function body will reflect on the actual argument variable.

Program to demonstrate call by reference mechanism

```
#include<stdlib.h>
#include<iostream.h>
#include<iostream.h>
void read(int &x);// prototype type for read function.
void main()
{
int a,b;
cout <<"enter two integers:";
read(a);
```

```
read(b);
cout<<"a is " <<a<<end1;
cout<<"b is"<<endl;
}
void read(int&x)
{
cin >>x;
}
```

output:

```
enter two integers:3
3
a is 3
b is
```

Program to read two variable and arrange them in order using 3 functions read (), swap (), display().

```
#include<iostream.h>
#include<conio.h>
void read(int &x,int &y);
void swap(int &x,int &y);
void show result(int x,int y);
void main()
:
int a,b;
cout <<”enter a&b values :”;
read (a,b);
if(a>b)
swap(a,b);
showresult(a,b);
getch();
}
void read (int &x, int &y)
{
cin>>x>>y;
return;
}
void swap(int &x,int &y)
{
x=x+y;
y=x-y;
x=x-y;
return;
}
void show (int x, int y)
```

```
{
cout<<"after swaping:"
<<a<< end <<b;
return;
}
```

output:

```
enter a&b values :3 4
after swaping: 4 3
```

3.4 Function Overloading

Writing two or more function definitions with the same function name is called function overloading. Such overloading definitions must have different number of formal parameters or it can have same number of parameters with different data types. When there is a function call, the compiler uses the function definition whose number of formal parameters and the type of parameters match the argument in the function call.

C++ allow us to use the function name by overloading them .However, it is not possible to overload a function name by giving two functions that differ only in the type of return value. I.e., the overloaded definitions are uniquely identified with the help of arguments list for the functions.

3.5 Void Functions

Sub tasks are implemented as functions in C++. A subtasks might produce several values are it might return no value. A function must either return a single value or return no value at all. A function that return no value is called a void function. In the void function it does not return any value to the rest of the program but produces the output on the screen.

Syntax:

```
//Function prototype
void function name(parameter list);
//function definition.
Void function name(parameter list)
{
body of the function;
```

```
:
:
return;
}
```

Program to demonstrate void functions

```
#include<iostream.h>
void starline(void) // this prints line of 10 *'s
{
for ( int  i=1; i<=10;i++)
cout<<"*";
return;
}
void starline(int n)
{
for (int i=1;i<=n;i++)
cout<<"*";
return;
}

#include<iostream.h>
void main()
{
starline();
int n;
cout<<"enter n value";
cin>>n;
starline(n);
}
```

output:

```
enter n value20
```

Void function may not have any parameters at all. It is optional to write a return statement in your void function. However, a return statement is used to make a conditional exit.

```
void print quotient(int x, int y);
{
if (y==0)
return;
cout <<"quotient="<<x/y;
}
```

3.6 Function Calling Another Function

A function body can contain a call to another function. When this inner function is called its prototype should appear before its first use. Although the function call is included in the definition of another function, the definition of the called function should be put outside the main program(definition).

e.g:

```
void order (int & n1, int&n2)
{
        if (n1>n2)
        swap (n1,n2);
}
}
void swap (int n1, int &n2)
{
        int temp
        temp = n1;
        n1 = n2;
        n2 = temp;
        return;
}
```

3.7 Inline Functions

When a function is called lot of extra time is taken in jumping to the function saving registers pushing arguments into stack and returning into the calling function for very small function this process increases the overheads (time). To eliminate the cost of calls to small functions a new feature called in line function is proposed. An inline function is a function i.e. expanded in line when it is invoked. The compiler replaces the function call with the corresponding function code (similar to macro expansion)

Syntax:

```
inline function name(parameter list)
{
function body
}
```

3.8 Default Arguments

C++ allows a call to the function without specifying all its arguments. In such conditions, the function assigns a default value to the parameter which does not have a matching argument in the function call.

Default values are specified when the function is declared. The compiler looks at the prototype uses and allots the program for possible default values we must add default from right to left in proto typing default arguments are useful where some arguments always have the same value.

program to demonstrate the usage of default arguments

```
#include<iostream.h>
float value (float p, float t, float r=0.15);
void printline(char ch='*', int len=40);//Default arguments
int main()
{
float amount;
printline();
amount=value(5000,0.5);
cout<<"Amount = "<<amount;
return 0;
}
float value (float p, float t, float r)
{
float amount;
amount=(p*t*r)/100.0;
return amount;
}
void printline(char ch, int len)
{
for ( int i=0; i<=len;i++)
cout<<ch;
cout<<"\n";
}
```

output:

```
*********************************
Amount = 3.75
```

3.9 Recursive Function

A function that contains a function call to itself, or a function call to a second function which eventually calls the first function is known as a recursive function . The recursive definition for computing the factorial of number can be expressed as follows :

fact (n)= { 1 if n=0

N*fact (n-1), otherwise

Recursion, as the name suggests, revolves around a function recalling itself . The recursive approach of problem solving substitute the given problem with another problem of the same form in such a way that the new problem is simpler than the original .

Two important conditions which must be satisfied by any recursive function are :

1. Each time a function calls itself it must be nearer, in some sense, to a solution.
2. There must be a decision criterion for stopping the process or computation .

Recursive function involves saving the return address, formal parameters, local variable upon entry , and restore these parameters and variable on completion.

factorial of a number using recursion

```
#include<iostream.h>
void main (void)
{
      int n;
      long int fact(int);    //prototype
      cout<<"Enter the number whose factorial is to be found
";
      cin>>n;
       cout<<"The factorial of "<<n<<"is"<<fact(n)<<endl;
}
long fact( int num )
```

```
{
    if(num==0)
        return 1;
    else
        return num *fact (num -1);
}
```

output:

Enter the number whose factorial is to be found 5
The factorial of 5is120

3.10 Procedural Abstraction

The principle of procedural abstraction is that, the function should written so that it can be used like a black box. This means that the programmer who uses the function need not know the body of the function to see how the function works. The function prototype and its comment should be known in order to use the function.

How To Write A Black Box Function Definition : The prototype comment should tell the programmer all the condition that are require as argument the function and it should describe the value returned by the function when the function is called.All variables used in the function body should be declared in the function body itself (formal parameters need not be declared).

The function prototype is broken down into two kinds of information called a pre- condition and post- condition. Pre – condition states that the assumption to be true when the function is called .The function should not be used and can't be expected to perform correctly unless the pre-condition holds. The post-condition describes the effect of the function call. i.e.., what will be true after the function is executed, when the pre-condition holds.

```
e.g.: void swap(int & var1 , int& var2);
        // Pre-condition: var1, var2 has been given values
        // Post condition: The value of var1, var2 has been
                           changed
```

Note: Designing a function that can be used a black box can also be called as information hiding .

3.11 Testing And Debugging

In top-down design each function is designed, coded (C++ program) and tested as a separate unit from the rest of the program. The given task is divided into smaller and manageable subtask. These subtasks are converted into C++ functions. To test each of the functions independently, drivers and stubs are used.

A **driver** program is a temporary tool with minimum code. This program gets values from the function arguments in the simplest possible way. It need not do all the calculations that the final program performs.

This program contains some loop statements to repeat the test with different values. Sometimes it is not possible to test a function without using some other functions output which was not yet tested.

In this case a simplified version of the untested function called a **stub** is used. A stub need not perform correct necessary value for testing in the simplest possible way.

Rules And Methods Of Testing: Every function should be tested in a program. The common approach for testing basic functions like I-O is using the driver programs. We use stubs to test the remaining functions.

The stubs are replaced by functions one at a time, i.e., replace by its function def and tested once the function is fully tested another stub is replaced and this process continues until a final program is produced.

3.12 Summary

- The function is defined and the types of functions , predefined and programmer defined functions are discussed.
- Call by value and call by reference mechanism in parameter passing to functions are studied.
- Details of Function overloading, void functions are made clear.

- Calling a function within a function, passing default arguments to the functions are discussed.
- Procedural abstraction and writing stubs and driver programs while testing and debugging are covered.

3.13 Technical terms

Actual Arguments: Arguments that are used in the function call.

Formal Arguments: Arguments that are used in function header. They are placeholders that is filled with function arguments when the function is called.

Function declaration: This provides the information for the function call. This is also known as function prototyping.

Function definition: It describes how the function computes the value it returns.

Function Overloading: A language feature that allows a function to be given more than one definition. The types of arguments with which the function is called, determines which definition is used.

Information Hiding: The hiding of the state and its implementing details in the module.

Inline functions: A function definition such that each call to the function is replaced by the statements that define the function.

3.14 Model Questions

1. Discuss the different types of functions?
2. What are predefined and programmer defined function? Explain with examples.
4. Explain the different mechanisms in parameter passing.
5. What are void functions? Explain with example.
6. What is an inline function? Explain with example.
7. What is a driver ?
8. What is a stub?
9. Explain the principle of procedural abstraction.

3.15 Reference Books

Object-oriented programming with C++

by E.Bala Gurusamy

Problem solving with C++

by Walter Savitch

Mastering C++

by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer
Dept. Of Computer Science
JKC College
GUNTUR.