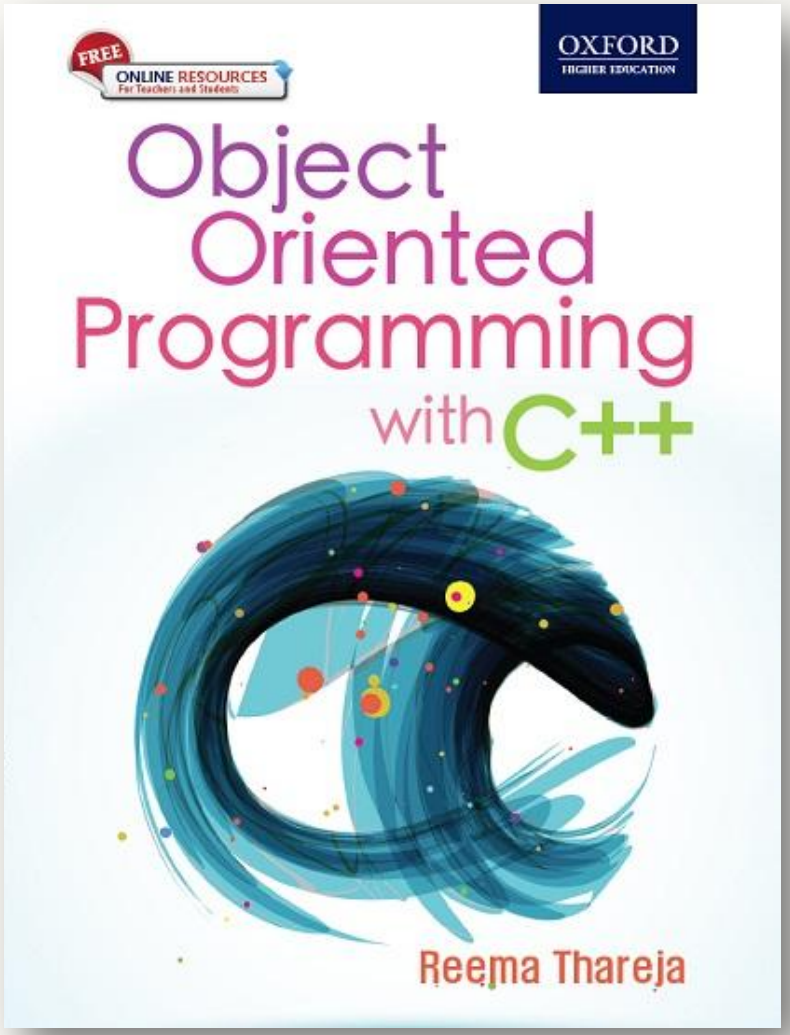


OXFORD
HIGHER EDUCATION

Object Oriented Programming with C++

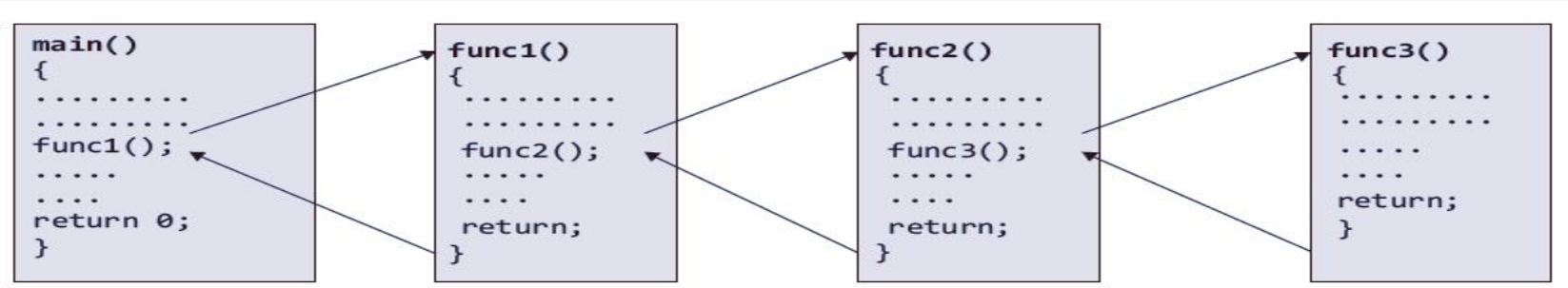
Reema Thareja

Chapter Four

Functions

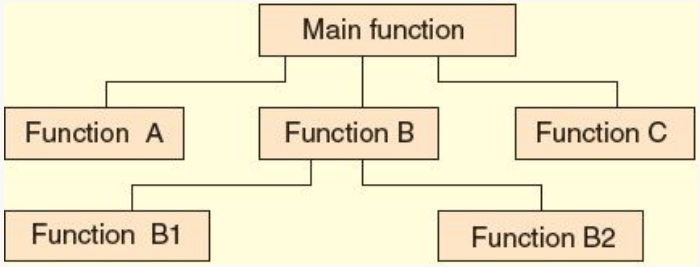
INTRODUCTION

- C++ enables programmers to break up a program into segments commonly known as functions.
- Each function can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task.
- It is not necessary that the main() can call only one function. It can call as many functions as it wants and as many times as it wants.
- Any function can call any other function



NEED FOR FUNCTIONS

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately.
- Understanding, coding, and testing multiple separate functions are far easier than doing the same for one huge function.
- All libraries in C++ contain a set of functions that programmers are free to use in their programs.These functions have been prewritten and pretested.
- When a big program is broken into comparatively smaller functions, different programmers working on that project can divide the workload by writing different functions.
- Like C++ libraries, programmers can also make their functions and use them from different points in the main program or any other program that needs its functionalities.



USING FUNCTIONS

- A function, f , that uses another function g, is known as the calling function and g is known as the called function.
- The inputs that the function takes are known as arguments or parameters.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function.
- If the called function accepts the arguments, the calling function will pass parameters, otherwise, it will not do so.
- Function declaration is a declaration statement that identifies a function with its name, a list of arguments that it accepts, and the type of data it returns.

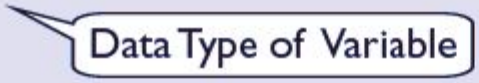
USING FUNCTIONS

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

FUNCTION DECLARATION OR FUNCTION PROTOTYPE

```
return_data_type function_name(data_type variable1, data_type variable2,..);
```

- function_name is a valid name for the function.
- return_data_type specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.
- data_type variable1, data_type variable2, … is a list of variables of specified data types.
- After the declaration of every function, there is a semicolon.
- The name of the function is global.

Function declaration	Use of the function
`char convert_to_uppercase (char ch);` 	Converts a character to upper case. The function receives a character as an argument, converts it into upper case, and returns the converted character back to the calling program.
`float avg (int a, int b);` 	Calculates average of two numbers a and b received as arguments. The function returns a floating point value.
`int find_largest (int a, int b, int c);` 	Finds the largest of three numbers a, b, and c received as arguments. An integer value which is the largest number of the three numbers is returned to the calling function.
`double multiply(float a, float b);` 	Multiplies two floating point numbers a and b that are received as arguments and returns a double value.
`void swap (int a, int b);`	Swaps or interchanges the value of integer variables a and b received as arguments. The function returns no value; therefore, the data type is void.
`void print(void);`	The function is used to print information on screen. The function neither accepts any value as argument nor returns any value. Therefore, the return type is void and the argument list contains void.

FUNCTION DEFINITION

The syntax of a function definition can be given as

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{ ……….
  statements
  ……….
  return( variable);
}
```

While `return_data_type function_name(data_type variable1, data_type variable2,..)` are known as the function header, the rest of the portion comprising program statements within `{ }` is the function body which contains the code to perform the specific task.

The function header is same as that of function declaration. The only difference between the two terms is that a function header is not followed by a semicolon. The list of variables in the function header is known as the formal parameter list. The parameter list may have zero or more parameters of any data type. The function body contains instructions to perform the desired computation in a function.

FUNCTION CALL

A function call statement has the following syntax:

```
function_name(variable1, variable2, …);
```

- Function name and the number and type of arguments in the function call must be the same as that given in the function declaration and function header of the function definition.
- If, by mistake, the parameters passed to a function are more than specified to accept, then the extra arguments will be discarded.
- If, by mistake, the parameters passed to a function are less than specified to accept, then the unmatched argument will be initialized to some garbage value (i.e. arbitrary meaningless value).
- Names, and not the types, of variables in function declaration, function call, and header of function definition may vary.
- Arguments may be passed in the form of expressions to the called function.

Example 4.1 To add two integers using functions

```
#include<iostream.h>
int sum(int a, int b);   // FUNCTION DECLARATION
int main()
{   int num1, num2, total = 0;
    cout<<"\n Enter two numbers : ");
    cin>>num1>>num2;
    total = sum(num1, num2);      // FUNCTION CALL
    cout<<"\n Total = "<<total;
}
// FUNCTION DEFNITION
int sum ( int a, int b)      // FUNCTION HEADER
{          // FUNCTION BODY
    int result;
    result = a + b;
    return result;
}
```

OUTPUT

```
Enter the two numbers : 20   30
Total = 50
```

RETURN STATEMENT

- The return statement is used to terminate the execution of a function and return control to the calling function.
- When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- A return statement may or may not return a value to the calling function.

```
return <expression>;
```

Using more than one `return` statement

```
#include<iostream.h>
int check_relation(int a, int b);       // FUNCTION DECALARATION
main()
{   int a = 3, b = 5, res;
    res = check_relation(a, b);         // FUNCTION CALL
    if(res == 0)                        // Test the returned valu
        cout<<"\n EQUAL";
    if(res == 1)
        cout<<"\n a is greater than b";
    if(res == -1)
        cout<<"\n a is less than b";
}
int check_relation(int a, int b)        // FUNCTION DEFINITION
{   if(a == b)
    return 0;
if(a > b)
    return 1;
else if (a < b)
    return -1;
}
```

OUTPUT

```
a is less than b
```

PASSING PARAMETERS TO THE FUNCTION

- When a function is called, the calling function may have to pass some values to the called function.
- There are two ways in which arguments or parameters can be passed to the called function.
- They include the following:
  - Call-by-value, in which values of the variables are passed by the calling function to the called function. The programs that we have written so far call the function using call-by-value method of passing parameters.
  - Call-by-address, in which the address of the variables is passed by the calling function to the called function.
  - Call-by-reference , in which a reference of the variable is passed by the calling function to the called function.

Call-by-Value

- In this method, the called function creates new variables to store the value of the arguments passed to it.
- Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function.
- In the calling function, no change will be made to the value of the variables.

```
#include<iostream.h>
void add(int n);  // FUNCTION DECLARATION
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(num);  // FUNCTION CALL
    cout<<"\n The value of num after calling the function = "<<num;
}
void add(int n)   // FUNCTION DEFINITION
{   n = n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

OUTPUT

```
The value of num before calling the function = 2
The value of num in the called function = 20
The value of num after calling the function = 2
```

Call-by-Address

- The call-by-pointer or call-by-address method of passing arguments to a function copies the address of an argument into the formal parameter.
- The function then uses the address to access the actual argument. This means that any changes made to the value stored at a particular address will be reflected in the calling function also.
- The called function uses pointers to access the data. In other words, the dereferencing operator is used to access the variables in the called function.

```
#include<iostream.h>
void add( int *n);
int main()
{   int num = 2;
    cout<<"\n The value of num before calling the function = "<<num;
    add(&num);
    cout<<"\n The value of num after calling the function = "<<num;
}
void add( int *n)
{   *n = *n + 10;
    cout<<"\n The value of num in the called function = "<<n;
}
```

OUTPUT

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

Call-by-Reference

- When the calling function passes arguments to the called function using call-by-value method, the only way to return the modified value of the argument to the caller is by using the return statement explicitly.
- A better option when a function can modify the value of the argument is to pass arguments using call-by-reference technique.
- In call-by-reference, we declare the function parameters as references rather than normal variables.
- When this is done, any changes made by the function to the arguments it received are visible by the calling program.

```
#include<iostream>
using namespacestd;
void add(int &n);
int main()
{
 int num=2;
 cout<<"Hello World";
add(num);
cout<<num;
return 0;
}
void add(int &p)
{
p=p+2;
}
```

DEFAULT ARGUMENTS

- When a function is specified with default arguments, the function can be called with missing arguments.
- When a function is called with a missing argument, the function assigns a default value to the parameter.
- This default value is specified by the programmer in the function declaration statement.
- While specifying the default values during function declaration, the programmer must keep the following in mind.
- Only trailing arguments can have default values; therefore, specify them from right to left.
- No argument specified in the middle can have default values.

```
int my_func(int a, int b, int c = 10);                  //Correct
int my_func(int a, int b = 5, int c = 10);              //Correct
int my_func(int a = 1, int b = 5, int c = 10);          //Correct
int my_func(int a, int b = 5, int c = 10);              //Wrong
```

Programming Tip: Arguments explicitly specified in function call override the default values given during function declaration.

Program 4.9 Write a program to calculate the volume of a cuboid using default arguments.

```
#include<iostream.h>
int Volume( int length, int width = 3, int height = 4 );
int main()
{ cout<<"\n Volume = "<<Volume(4, 6, 2);
  cout<<"\n\n Volume = "<<Volume(4, 6);
  cout<<"\n\n Volume = "<<Volume(4);
}
int Volume(intlength, int width, int height)
{      cout<<"\n Length = "<<length<<" Width = "<<width " and Height
       = "<<height;
    return length * width * height;
}
```

OUTPUT

```
Length = 4 Width = 6 and Height = 2
Volume = 48
Length = 4 Width = 6 and Height = 4
Volume = 96
Length = 4 Width = 3 and Height = 4
Volume = 48
```

RETURN BY REFERENCE

- This allows a function to be used on the left side of an assignment statement.

The following are the key points to remember while returning by reference:

- The function should not return a local variable by reference as it will go out of scope immediately as soon as the function ends. This means the following code will generate a compiler error.

```
int &my_func(int num)
{   int x = num*2;
        return x;      //ERROR, x is a local variable
}
```

- The function may return a reference to a static variable. Therefore, the code given here is permissible in C++.

```
int &my_func()
{   static int x;
    return x;
}
```

- Variables passed by reference can be returned by reference.
- Return by reference is extensively used to return structure variables and objects of classes.

```
#include<iostream.h>
int &greater(int &x, int &y)
{   if(x>y)
        return x;
    else
        return y;
}
main()
{   int num1, num2, large;
    cout<<"\n Enter two numbers : ";
    cin>>num1>>num2;
    cout<<"\n Two numbers are : "<<num1<<" "<<num2;
    large = greater(num1, num2);
    cout<<"\n Large = "<<large;
    greater(num1,num2) = -1;
    cout<<"\n Two numbers are : “<<num1<<" "<<num2;
}
```

OUTPUT

```
Enter two numbers : 5 2
Two numbers are : 5 2
Large = 5
Two numbers are : -1 2
```

PASSING CONSTANTS AS ARGUMENTS

- When parameters are passed by reference parameter, the called function may intentionally or inadvertently modify the actual parameters.
- However, at times, the programmer may strictly want the actual parameters not to be modified by the called function.
- In such cases, a constant parameter must be passed.
- Therefore, using the const keyword allows programmers to achieve performance benefits while ensuring that the actual parameter is not modified.

```
#include<iostream.h>
void add_2(int const &x)
{   x = x + 2;   // ERROR, cannot modify a constant
    cout<<"\n The numbers is now : "<<x;
}
main()
{   int num1;
    cout<<"\n Enter a number : ";
    cin>>num1;
    add_2(num1);
}
```

OUTPUT

```
Error
```

VARIABLES SCOPE

- In C++, all constants and variables have a defined scope.
- By scope, we mean the accessibility and visibility of variables at different points in the program.
- A variable or a constant in C++ has four types of scope—block, function, file, and program scope.

Block Scope

- A statement block is a group of statements enclosed within an opening and closing curly brackets ({ }).
- If a variable is declared within a statement block, as soon as the control exits that block, the variable will cease to exist.
- Such a variable also known as a local variable is said to have a block scope.

Example 4.10 Code which illustrates the block scope concept

```
#include <iostream.h>
int main()
{   int x = 10;
    int i=0;
    cout<< "\n The value of x outside the while loop is "<<x;
    while (i<3)
    {
        int x = i;
        cout<<"\n The value of x inside the while loop is "<<x;
    i++;
    }
    cout<<"\n The value of x outside the while loop is "<< x;
}
```

OUTPUT

```
The value of x outside the while loop is 10
The value of x inside the while loop is 0
The value of x inside the while loop is 1
The value of x inside the while loop is 2
The value of x outside the while loop is 10
```

VARIABLES SCOPE

Function Scope

- Function scope indicates that a variable is active and visible from the beginning to the end of a function.
- In C++, only the goto label has function scope.

Scope of the Program

- If you want the functions to be able to access some variables which are not passed to them as arguments, declare those variables outside any function blocks. Such variables are commonly known as global variables.
- Global variables are those variables that can be accessed from any point in the program.

VARIABLES SCOPE

File Scope

- When a global variable is accessible until the end of the file, the variable is said to have file scope.
- To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type as follows:

```
static int x = 10;
```

- A global static variable can be used anywhere from the file in which it is declared but it is not accessible by any other files. Such variables are useful when the programmer writes his own header files.

Example 4.11 Code to illustrate the concept of program scope

```
#include<iostream.h>
int x = 10;
void print();
int main()
{   cout<<"\n The value of x in the main() = "<<x;
    int x = 2;
    cout<<"\n The value of local variable x in the main() = "<<x;
    print();
}
void print()
{   cout<<"\n The value of x in the print() = "<<x;

}
```

OUTPUT

```
The value of x in the main() = 10
The value of local variable x in the main() = 2
The value of x in the print() = 10
```

STORAGE CLASSES

- The storage class of a variable defines the scope or visibility and life time of variables and/or functions declared within a C++ program.

```
<storage_class_specifier> <data type> <variable name>
```

Feature	Storage class			
	Auto	Extern	Register	Static
Accessibility	Accessible within the function or block in which it is declared	Accessible within all program files that are a part of the program	Accessible within the function or block in which it is declared	Local Accessible within the function or block in which it is declared Global: Accessible within the program in which it is declared
Storage	Main memory	Main memory	CPU register	Main memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Exists throughout the execution of the program	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Local Retains value between function calls or block entries Global Preserves value in program files
Default value	Garbage	Zero	Garbage	Zero

INLINE FUNCTIONS

- The major difference between an ordinary function and an inline function is that when an inline function is called, the compiler places a copy of its code at each point of call.
- As in case of an ordinary function, the compiler does not have to jump to the called function.
- This saves the function call overhead and results in faster execution of the code.
- We can make a function inline by taking caring of two aspects as follows:
  - First, write the keyword inline before the function name.
  - Second, define that function before any calls are made to it.

INLINE FUNCTIONS

- However, the programmer must not forget that keyword inline just makes a request to the compiler to make the function inline (and place its code at each point of call).
- The compiler may ignore the request if the function has too many lines.

Example 4.16 To find the greater number using an inline function

Programming Tip: `main()` cannot be used as an inline function.

```
#include<iostream.h>
inline int greater(int x, int y) // Function definition
{ return (x > y)? x : y; // Function returning greater of the two nos
}
int main( )
{    int num1, num2;
   cout << "\n Enter two numbers : ";
cin>>num1>>num2;
   cout<<"\n The greater number is : "<<greater(num1,num2);       // Function call
}
```

OUTPUT

```
Enter two numbers : 5 9
The greater number is : 9
```

ADVANTAGES

- An inline function generates faster code as it saves the time required to execute function calls.
- Small inline functions (three lines or less) create less code than the equivalent function call as the compiler does not have to generate code to handle function arguments and a return value.
- Inline functions are subject to code optimizations that are usually not available to normal functions as the compiler does not perform inter-procedural optimizations.
- Inline function avoids function call overhead. As a result, we need to save variables and other program parameters on the system stack.

ADVANTAGES

- Since there are no function calls, the overhead of returning from a function is also avoided.
- It allows the compiler to apply intra-procedural optimization

DISADVANTAGES

- Size of the program increases.
- Large programs may take longer time to be executed.
- At times, the program may not fit in the cache memory.
- There may be a problem in making efficient use of CPU registers if the inline function has many register variables.
- If the code of the inline function is modified, the entire program needs to be re-compiled.
- Inline functions should not be used for designing embedded systems due to memory size constraints.

COMPARISON OF INLINE FUNCTIONS WITH MACROS

- Macro invocations skip the job of type checking; this is a must-to-do work in function calls.
- Macros cannot return a value while a function can return a value.
- Macros use mere textual substitution which can give unintended results due to inaccurate reevaluation of arguments and order of operations.
- Debugging of compiler errors in case of macros is more difficult than debugging functions.
- All constructs cannot be expressed using macros; however, with functions, they can be expressed with ease.
- Macros have a slightly difficult syntax while the syntax of writing function is similar to that of a normal function.

FUNCTION OVERLOADING

- Function overloading, also known as method overloading, is a feature in C++ that allows creation of several methods with the same name but with different parameters.
- For example, print(), print(int), and print("Hello") are overloaded methods.
- While calling print() , no arguments are passed to the function; however, when calling print(int) and print("Hello") , an integer and a string arguments are passed to the called function.
- Function overloading is a type of polymorphism.

FUNCTION OVERLOADING

- Basically, there are two types of polymorphism:
- Compile time (or static) polymorphism and run-time (or dynamic) polymorphism.
- Function overloading falls in the category of static polymorphism which calls a function using the best match technique or overload resolution.

MATCHING FUNCTION CALLS WITH OVERLOADED FUNCTIONS

When an overloaded function is called, one of the following cases occurs:

- Case 1: A direct match is found, and there is no confusion in calling the appropriate overloaded function.
- Case 2: If a match is not found, a linker error will be generated. However, if a direct match is not found, then, at first, the compiler will try to find a match through the type conversion or type casting.
- Case 3: If an ambiguous match is found, that is, when the arguments match more than one overloaded function, a compiler error will be generated. This usually happens because all standard conversions are treated equal.

```
void print(int);   // Function declaration
print('R');      // Function call
```

Example 4.19 To demonstrate the ambiguity in function call

```
#include<iostream.h>
void print(int n){ cout<<n;}
void print(char c){ cout<<c; }
void print(float f){ cout<<f;}
main()
{    print(5);   //Function call
     print (98.7);
     print('R');
}
```

OUTPUT

```
Error
```

Example 4.20 To compute the volume of different shapes using function overloading concept

```
#include<iostream.h>
int volume(int side)
{   return side*side*side; // cube
}
float volume(float radius, float height)
{   return 3.14+radius*radius*height; //cylinder
}
long int volume(int length, int breadth, int height)
{   return length*breadth*height; //cuboid
}
main()
{  int s,l,b,height;
   float r, h;
   cout<<"\n Enter the side of the cube : ";
   cin>>s;
   cout<<"\n Volume of cube with side "<<s<<" = "<<volume(s);
   cout<<"\n \n\n Enter the radius and height of the cylinder : ";
   cin>>r>>h;
   cout<<"\n Volume of cylinder with radius "<<r<<" and height "<<h<<" = "<<volume(r,h);
   cout<<"\n Enter the length, breadth and height of the cuboid : ";
   cin>>l>>b>>height;
   cout<<"\n Volume of cuboid with length "<<l<<" breadth "<<b<<" and height
   "<<height<<" = "<<volume(l,b,height);
}
```

OUTPUT

```
Enter the side of the cube : 3
Volume of cube with side 3 = 27
Enter the radius and height of the cylinder : 3 4
Volume of cylinder with radius 3 and height 4 = 39.13
Enter the length, breadth and height of the cuboid : 2 3 4";
Volume of cuboid with length 2 breadth 3 and height 4 = 24
```

Functions that Cannot be Overloaded

- Functions that differ only in the return type. For example, the program given here will give compile time error.

```
#include<iostream.h>
int my_func()     { return 1; }
char my_func()    { return 'E'; }
main()
{ int num = my_func();
 char c = my_func();
}
```

- Parameter declarations that differ only in a pointer * versus an array [] are equivalent. A program with the following declarations will give a compilation error as both declarations are equivalent. The reason why they are equivalent will be clear in the chapter on Pointers.

```
int my_func(int *ptr);
int my_func(int ptr[]);
```

- If parameters in two functions differ only in the presence or absence of const and/or volatile, then those functions are considered to be equivalent. For example, the program having following declarations will not compile.

```
int my_func(int n);
int my_func(const int n;
```

- Using typedef does not introduce a new type, therefore, the following two function declarations are equivalent and this cannot be overloaded. (Use of `typedef` will be studied in chapter on Structures).

```
typedef int integer;
int my_func(int n);
int my_func(integer n;
```

- Function declarations that differ only in their default arguments are equivalent. Therefore, the program having function declarations given below will not compile.

```
int my_func(int n);
int my_func(int n = 10);
```

- Function declarations that differ only in a reference parameter and a normal parameter cannot be overloaded. Therefore, the program having function declarations given here will not compile.

```
int my_func(int n);
int my_func(int &n);
```

RECURSIVE FUNCTIONS

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases which are given as follows:
- Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- Recursive case, in which first the problem at hand is divided into simpler sub parts.
- Second, the function calls itself but with sub parts of the problem obtained in the first step.
- Third, the result is obtained by combining the solutions of simpler sub-parts.

```
        Problem                          Solution
        5!                               5 X 4 X 3 X 2 X 1!
=       5 X 4!                   =       5 X 4 X 3 X 2 X 1
=       5 X 4 X 3!               =       5 X 4 X 3 X 2
=       5 X 4 X 3 X 2!           =       5 X 4 X 6
=       5 X 4 X 3 X 2 X 1!       =       5 X 24
                                 =       120
```

Example 4.21 To calculate the factorial of a number recursively

Programming Tip: Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls, thereby resulting in an error condition known as an infinite stack.

```
#include<iostream.h>
int Fact(int);   // FUNCTION DECLARATION
main()
{ int num;
  cout<<"\n Enter the number : ";
  cin>>num;
  cout<<"\n Factorial of "<<num<<" = "<<Fact(num);// FUNCTION CALL
}
int Fact( int n)   // FUNCTION DEFINITION
{   if(n==1)
      retrurn 1;
   return (n * Fact(n-1));   // FUNCTION CALL
}
```

OUTPUT

```
Enter the number : 5
Factorial of 5 = 120
```

ADVANTAGES

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion represents like the original formula to solve a problem.
- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

DISADVANTAGES

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack.
- If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counter part.
- It is difficult to find bugs, particularly, when using global variables

FUNCTIONS WITH VARIABLE NUMBER OF ARGUMENTS

- In case we need a function that should accept a variable number of arguments, we have to design a function that accepts the data-type and/or number of arguments at the run-time while execution and not during compilation.
- Let us take a sample program that accepts any number of values and then return the average.
- To write such a program, we must include the cstdarg header file to use the following macros:
- va_list that stores the list of arguments.
- va_start to initialize the list.
- It is a macro from which we can begin reading arguments from it. It accepts two arguments— va_list and a last named

FUNCTIONS WITH VARIABLE NUMBER OF ARGUMENTS

argument. After the last named argument, the number of arguments read is stored.

- va_arg returns the next argument in the list. It helps in accessing the individual arguments
- in the list. For this, you must just need to specify the type of argument to retrieve.
- va_end is used to clean up the variable argument list once we are done with it.
- A function that is supposed to accept variable number of arguments must be declared in a special way.
- Instead of writing the last argument, you must put an ellipsis (like '…'). Therefore, int my_func(int x, …); means that my_func accepts any number of arguments.

Example 4.22 Using variable number of arguments

```
#include<stdarg.h>
#include<iostream.h>
float avg(int argc, …)
{   va_list args;
    int i, sum = 0;
    va_start(args, argc);
    for(i = 0;i < argc;i++)
       sum += va_arg(args, int); // Adds next value in args to sum
    va_end(args);
    return( (float) sum/argc);
}
main()
{   float result;
   result = avg(7,8,9,10,4,5,10,10,10,9,7,8,9,10,10);
   cout.precision(2);
   cout<<"\n AVERAGE = "<<result;
}
```

OUTPUT

```
AVERAGE = 8.00
```