Chapter 1

Graduating from C to C++

Learning Outcomes

After reading this chapter, students will have the ability to

- explain the new style of commenting, new data types
- explain the enhanced approach of handling of structures and unions
- explain the relaxation in the place of declaration of variables
- explain the concept of reference variables
- explain the new style of typecasting, new cast operators, new operator keywords, and new headers
- explain the concept of namespaces
- explain the parameter passing by reference
- explain the concept of stream based I/O
- explain the concept of inline functions and its comparison with macros
- explain the enhancements of structures and unions to include functions as its members
- explain the concept of default arguments
- explain the concept of function overloading
- explain the concept of function templates
- memory management operators

1.1 INTRODUCTION

The C++ language is a very powerful general-purpose programming language that supports procedural programming as well as object-oriented programming. It was developed by **Bjarne Stroustrup** at AT&T Bell Laboratories, New Jersey, USA, in the early 1980's. He found that as the magnitude of problem size and its complexity grows, it becomes extremely difficult to manage it, using most of procedural languages, including C language. He was a strong admirer of Simula67 and C languages, and wanted to have a language that combines the best of both of the languages, *i.e.*, a language that supports object oriented programming, and at the same time have the power and elegance of C language. The outcome of his efforts ultimately leads to the development of C++ language. Since classes were a major addition to the original C language, he initially called the new language **'C with Classes'**. Later on, the name was changed to C++ in the year 1983. The idea of suffixing C with ++ came from the increment operator as the new features were added to the C language which were in use since long.

During the early 1990's, the language underwent a number of changes and improvements. In the year 1997, the ANSI standards committee standardized these changes, and added several new features to the language specifications.

The C++ language is treated as super set of C language because the developer of C++ language has retained all features of C, enhanced some of the existing features, and incorporated new features to support for object-oriented programming.

In this chapter, we will discuss about the enhanced features of C++ language over C language. Every C programmer can very easily grasp these features without bothering about object-oriented features of C++ language.

1.2 NEW STYLE OF COMMENTING

The C++ language supports new style of commenting. It starts with two consecutive forward slashes '//'. This style of comments can appear on a separate line or following any instructions. The following are the examples of valid C++ comments:

```
// This is a C++ style comment.
int rno;  // variable for roll number
```

The only limitation of this style of commenting is that comments are limited to single line. There if you want to add multiple lines of comments, then every line of comment must be preceded by //.

As a guideline, whenever we want to add multiple line comments, we prefer C style commenting and for single line comments we prefer C++ style commenting. In the book we will be following these guidelines.

Listing 1.1

```
/*
 * Program to convert temperature from Centigrade scale to
 * Fahrenheit scale. This program also illustrates the use
 * of comments in a C++ program
*/
#include <iostream.h>
#include <iomanip.h>
void main()
{
    float fahrenheit, centigrade;   // declaration of variables

    cout << "Enter temperature in Celsius scale : ";
    cin >> centigrade;              // keyboard input

    fahrenheit = 1.8 * centigrade + 32;
    cout << "Equivalent Temperature in Fahrenheit = ";
    cout << fahrenheit << endl;     // output computed temperature
}
```

1.3 NEW DATA TYPES

New standard of C++ language supports following new data types:

- The *bool* data type
- The *wchar_t* Data Type

Note that the old C++ compilers don't support these data types.

1.3.1 The *bool* Data Type

The *bool* data type has been added to hold a boolean value — *true* or *false*. The boolean values *true* or *false* have been declared as keywords. A *bool* value occupies 1 bytes of memory. Boolean value *false* is internally represented as integer value 0 whereas boolean value *true* is internally represented as integer value 1. Following statements show declarations and use of *bool* type variables.

```
bool b1, b2;           // declares variable of type bool

b1 = true;             // assigns boolean value true
b2 = 0;                // assign value 0, i.e. false
bool b3 = false;       // declares and initialize with Boolean
                       // value false
```

You can assign any non-zero value to a *bool* variable to represent boolean value true, however, it is always converted to value 1 before assignment.

When the display the values of bool variables, value 0 will be printed for boolean value false and 1 for boolean value true.

```
bool b1 = false, b2 = true;
cout << "\nb1 = " << b1;
cout << "\nb2 = " << b2;
```

The output will be as follows:

```
b1 = 0
b2 = 1
```

However, if you want the value of *bool* variable to be printed as boolean value, i.e., as *false* and *true*, then the *boolalpha* manipulator can be used to accomplish this task.

This is illustrated below:

```
bool test = true;
cout << boolalpha;
cout << "test = " << test;
```

The output will be as follows:

```
test = true
```

A *bool* variable and boolean value can be used in an arithmetic expression. For example, the following statements

```
int x, y = 10;
bool f = false;
x = true + 2 * y - f;
```

are valid.

1.3.2 The *wchar_t* Data Type

The new character data type named *wchar_t* is added to hold 16-bit wide characters called *unicode characters*. The 16-bit characters are used to represent the character sets of other languages such as Spanish, Japanese, French, etc. The use of *wchar_t* is particularly useful if we are writing programs for international distribution.

A new character literal known as *wide_character* is also added and it uses 2-bytes of memory. Wide_character literals begin with the letter L, as shown below:

```
L‘xy’      // wide_character literal
```

1.4 PROMOTION OF USER-DEFINED DATA TYPES

In C++ language, when you create a user-defined data type, the tag of the user-defined data type acts as a data type like basic data type and you can use tag to declare a variable.

Consider the following example:

```
struct EMPLOYEE
{
    int   code;
    char  name[20];
    int   deptCode;
    float salary;
};
```

The statement

```
EMPLOYEE aEmployee;
```

declaring the structure variable *aEmployee*. Here tag *EMPLOYEE* acts as a data type. To accomplish the same thing in C, we have to use the statement

```
struct EMPLOYEE aEmployee;
```

In C language, if variables of same user-defined data type are required at many parts of the program, then we have to repeatedly use the keyword for appropriate user-defined data type before the tag, which may be cumbersome and many a times becomes the source of errors.

Therefore, to overcome this difficulty in C language, when we create a new user-defined data type, and assign a symbolic name to it using *typedef* statement as illustrated below:

```
typedef struct
{
    int   code;
    char  name[20];
    int   deptCode;
    float salary;
} EMPLOYEE;
```

Now the statements of type

```
EMPLOYEE aEmployee;
```

will work in C language.

Similar treatment is with *union* and *enum* user-defined data types.

Though the keyword *typedef* is retained in C++ language, but it is not used much.

1.5 VARIABLE DECLARATION

In C language, local variables can only be declared at the top of the function or at the beginning of the nested block. While coding the program, it may not be known in advance the number and type variables to be used. As and when a new variable is required, we have to move to the top of the function definition and declare the variable. Many a times we use the variable and forget to declare it that leads to syntax error.

However, in C++ language, this restriction is relaxed. A variable in C++ language can be declared anywhere in the function, but in such a case the scope of the variable will be limited to part of the function following its declaration. Hence, most of the C++ programmers follow the practice of declaring the variables at its first use.

Listing 1.2

```
/*
 *  Program to demonstrate the declaration of variables
 *  at the point of its first use
*/
#include <iostream.h>
#include <iomanip.h>
int main()
{
    // variable 'i' cannot be accessed before 'for' statement
    for ( int i = 0; i < 10; i++ )
    {
       cout << i << endl;
    }
    // variable 'i' can be accessed after 'for' statement
    cout << i << endl;
    return 0;
}
```

On other side, the declaration of variables at any position in the program reduces the readability of the program. Therefore, to enhance the readability of the program, it is better to declare the variables at the beginning of the function.

1.6 REFERENCE VARIABLES – *Variable Aliases*

In C language, we have used *value variables* and *pointer variables*. The value variables hold some value whereas the pointer variables are used to hold the address of some value variable. The C++ language supports one more type of variable called *reference variable*. A reference variable acts as an alias (alternative name) for other variable. Reference variables enjoy the simplicity of value variables and power of pointer variables.

A reference variable is bound to a value/pointer variable only at the point of its declarations using following syntax

```
DataType &ReferenceVariable = Variable;
```

The following examples illustrate the binding of reference variables.

```
char &ch1 = ch;  // ch1 is an alias of ch

int &x = y[100]; // x is an alias of y[100]

int n = 20;
int *p = &n;
int &m = *p;            // m an alias of *p and refers to n
```

Reference variables play important role while passing arguments to a function.

1.7 SCOPE RESOLUTION

The variables declared outside the functions are known as global variables. These variables are visible to all function following its declaration. But if there is a local variable with the same name in a function, there is no way to access the global variable in C language. In C++ language, a global variable can be accessed using *scope resolution operator* '**::**' (two consecutive colons).

The syntax of using the scope resolution operator is

```
::GlobalVariable
```

The next program illustrates the accessing of local and global variables with same name.

Listing 1.3

```
/*
 *  Program to demonstrate the accessing local and
 *  global variables with same name
*/

#include <iostream.h>
#include <iomanip.h>

int i = 10;       // declaration & definition of global variable

int main()
{
    int i=20;
    cout << "Local variable i = " << i << endl;
    cout << "Global variable i = " << ::i << endl;
    return 0;
}
```

1.8 NEW STYLE OF TYPECASTING

We know that basic data types are automatically converted to an appropriate type when used in assignments and expressions due to implicit type conversion facility provided by the language whereas this does not happen when the variable is passed as an argument during a function call. Similarly, this implicit type conversion facility does not work with user-defined data types. Through the use of explicit type conversion, (*the cast operator*), we can achieve the data type conversion for any sort of data type, *i.e.*, basic as well as user-defined.

In C language, the syntax for explicit type conversion is

```
(DataType) Variable/Expression
```

The data type is enclosed in parenthesis.

In C++ language, the syntax for explicit type conversion is

```
DataType(Variable/Expression)
```

The variable or expression is enclosed in parenthesis.

This looks very similar to a function call.

1.9 NEW CAST OPERATORS

We have used *cast operator*, also known as *cast* or *typecast*, where the data type is used to convert a value from one type to another type as shown below:

```
float a = 12.45;        // variable of type float
int x;                  // variable of type int
x = int(a);             // conversion from float to int
```

This type of conversion is necessary where automatic conversions are not possible.

Following new type of cast operators have been added:

- static_cast
- const_cast
- dynamic_cast
- reinterpret_cast

1.9.1 The *static_cast* Operator

Like the conventional cast operator, the *static_cast* operator is used for any standard conversion of data types.

It can also be used to cast a base class pointer into a derived class pointer.

Its syntax is

```
static_cast<type>(variable/expression)
```

where *type* is one of standard data types, and specifies data type of target variable.

```
float a = 12.45;              // variable of type float
int x;                        // variable of type int
x = static_cast<int>(a);      // conversion from float to int
```

A trivial question arises in mind – *why to use this new cast operator when the old style cast operators still works*? Note that the old style, which looks like a normal function call, got mixed up with rest of the code and therefore it is difficult to spot it.

This new cast operator is easy to spot and to search specially using automated tools.

1.9.2 The *const_cast* Operator

The *const_cast* operator is used for casting away *const* or *volatile*. Since the purpose of *const_cast* operator is to change its *const* or *volatile* attributes to normal attributes, the target variable should be of same type as that of source.

Its syntax is

```
const_cast<type>(variable)
```

where *type* is one of the standard data types.

1.9.3 The *dynamic_cast* Operator

Run-time type identification (RTTI) provides a means of determining an object's type at run time. Two operators are added for this purpose - *dynamic_cast* and *typeid*. We will discuss these operators one-by-one.

The *dynamic_cast* operator is used to cast the type of an object at run-time. Its main application is to perform casts on polymorphic objects. Basically it is used for downcasting from a base class pointer to a derived class pointer.

Its syntax is

```
dynamic_cast<type>(baseClassPtr)
```

where *type* is pointer to the derived class and *baseClassPtr* is a pointer of the base class. If the object held by the base class pointer is of derived class, then it will return the address of the derived class object otherwise returns NULL value, *i.e.*, value 0.

The *typeid* operator is used to obtain the type of the unknown objects such as their class names at run-time. Its syntax is

```
typeid(object).name()
```

For example, in the following statement

```
const char *objectType = typeid(object).name();
```

It will assign the type of object (class name) to pointer variable *objectType*. To do this, it uses member function *name()* of *type_info* class. Remember that to use *typeid* operator, you have to include header file *typeinfo.h* in your program.

1.9.4 The *reinterpret_cast* Operator

The *reinterpret_cast* operator is used for non-standard casts, *i.e.*, to change one type to fundamentally different type. It is oftenly used to cast one type of pointer to a different pointer type. It cannot be used for standard cast such as *float*, *int*, *double*, etc.

Its syntax is

```
reinterpret_cast<type>(sourcePtr)
```

where *type* is pointer to the target type and *sourcePtr* is a pointer of the source object. Consider the following example

```
float a = 12.45, *floatPtr;
int *intPtr;
floatPtr = &a;
intPtr = reinterpret_cast<int *>(floatPtr);
```

Here it will convert a pointer of type *float* to a pointer of type *int*.

The *const_cast* operator is used for casting away *const* or *volatile*. Since the purpose of *const_cast* operator is to change its *const* or *volatile* attributes to normal attributes, the target variable should be of same type as that of source. Its syntax is

```
const_cast<type>(variable)
```

where *type* is one of the standard data types.

1.10 OPERATOR KEYWORDS

New operator keywords have been added that can be used in place of several operators.

Table 1.1: Operator keywords

Operator	Operator Keyword	Description
&&	and	Logical AND
||	or	Logical OR
!	not	Logical NOT
!=	not_eq	Inequality
&	bitand	Bitwise AND
|	bitor	Bitwise inclusive OR
^	xor	Bitwise exclusive OR
~	compl	One's complement
&=	and_eq	Bitwise AND assignment
|=	or_eq	Bitwise inclusive OR assignment
^=	xor_eq	Bitwise exclusive OR assignment

Following statement

```
x < y && z != k
```

using operator keywords can be written as

```
x < y and z not_eq k
```

1.11 NEW HEADERS

A new way of specifying header files is incorporated. Here you don't have to suffix the header file name with *.h* extension. This style of header files works in conjunction with namespaces, which are described in next section.

```
#include <iostream>
#include <fstream>
```

The traditional style <*iostream.h*>, <*fstream.h*>, etc., is still fully supported.

1.12 NAMESPACES

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope may overlap (i.e. collide) with a variable of same name in a different scope, and thus creating a problem. Such overlapping can occur at many levels. This situation frequently occurs in third party libraries where same names are used for global identifiers such as functions.

To handle this problem, a new feature in the form of *namespace* is added. Each *namespace* defines a scope where all related identifiers are placed.

To use a *namespace member*, the member's name must be qualified with the *namespace* name and the scope resolution operator as shown below:

```
namespaceName::member
```

or a *using* statement must occur before the member is used. The *using* statements are generally placed in the beginning of the program. For example, the statement

```
using namespace namespaceName;
```

at the beginning of a program specifies that members of namespace *namespaceName* can be used in the program without preceding each member with the *namespaceName* and the scope resolution operator.

1.12.1 Defining a Namespace

We can define our own namespaces. The syntax for defining a namespace is

```
namespace namespaceName
{
   // declarations of variables, functions, and
   // classes
}
```

Note that there is no semicolon after the terminating brace.

1.12.2 Nesting of Namespaces

A namespace can be nested within another namespace as shown below:

```
namespace outerNamespaceName
{
   // . . .
   // . . .
   namespace innerNamespaceName
   {
       // . . .
       // . . .
       int m = 10;
   }
}
```

Variable *m* can be accessed using following ways:

```
cout << outerNamespaceName::innerNamespaceName::m;
```

OR

```
using outerNamespaceName;
cout << innerNamespaceName::m;
```

1.13 PARAMETER PASSING BY REFERENCE

In addition to *pass-by-value* and *pass-by-address*, parameters can also be *passed by reference* in C++ language. The parameter passed by reference has the functionality of pass-by-address and the syntax of pass-by-value. Any modification made through the formal reference parameter is also reflected in the actual parameter.

Consider the following implementation of a function to interchange values of two integer variables when the formal arguments are value variables.

Listing 1.4

```
/*
 * Program to demonstrate the argument passing using call by value
*/
#include <iostream.h>
#include <iomanip.h>
int main()
{
    int x = 10, y = 20;
    void swap ( int, int );  // function prototype
    cout << "Before call to swap function" << endl;
    cout << "Value of x = " << x << endl;
    cout << "Value of y = " << y << endl;
    swap (x, y);       // function call
    cout << "After call to swap function" << endl;
    cout << "Value of x = " << x << endl;
    cout << "Value of y = " << y << endl;
    return 0;
}
void swap ( int a, int b )
{
   int temp = a;
   a = b;
   b = temp;
}
```

In this case, the values of variables *x* and *y* are copied into formal parameters *a* and *b* respectively, which are value variables and acts as local variables to the function. The local copies of the variables are interchanged through another local value variable *temp*. The values of variables *x* and *y* are left unchanged.

Listing 1.5

```
/*
 * Program to demonstrate the argument passing
 * using call by address (pointer)
*/
#include <iostream.h>
#include <iomanip.h>
void main()
{
    int x = 10, y = 20;
    void swap ( int *, int * );    // function prototype
    cout << "Before call to swap function" << endl;
    cout << "Value of x = " << x << endl;
    cout << "Value of y = " << y << endl;
    swap (&x, &y);                 // function call
    cout << "After call to swap function" << endl;
    cout << "Value of x = " << x << endl;
    cout << "Value of y = " << y << endl;
    return 0;
}
void swap ( int *a, int *b )
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Now consider the implementation of *swap()* function to interchange values of two integer variables when the formal arguments are pointer variables.

```
void swap ( int *a, int *b )
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

The corresponding call to above function with *x* and *y* as actual parameters is

```
swap (&x, &y);
```

In this case, the addresses of variables *x* and *y* are copied into formal parameters *a* and *b* respectively, which are pointer variables and acts as local variables to the function. Here, values of the variables whose addresses are held in *a* and *b* are interchanged through local value variable *temp*. Here the values of variables *x* and *y* will be interchanged.

Listing 1.6

```
/*
 * Program to demonstrate the argument passing
 * using call by reference (alias)
*/
#include <iostream.h>
#include <iomanip.h>
int main()
{
    int x = 10, y = 20;
    void swap ( int &, int & );    // function prototype
    cout << "Before call to swap function" << endl;
    cout << "Value of x = " << x << endl;
    cout << "Value of y = " << y << endl;
    swap (x, y);                   // function call
    cout << "After call to swap function" << endl;
    cout << "Value of x = " << x << endl;
    cout << "Value of y = " << y << endl;
    return 0;
}
void swap ( int &a, int &b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

Now consider the implementation of *swap()* function to interchange values of two integer variables when the formal arguments are reference variables.

```
void swap ( int &a, int &b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

In the function prototype

```
void swap ( int &a, int &b );
```

the operator & indicates that formal parameters are of reference type, and they must be bound to the memory locations of actual arguments. Thus access made to these formal reference variable in the *swap()* function refers to the actual parameters. Here the values of variables *x* and *y* will be interchanged.

The call statement

```
swap ( x, y );
```

is translated into

```
swap ( &x, &y );
```

during the compilation process.

By comparing above versions of *swap()* functions, you will find that reference variables enjoy the simplicity of value variables and power of pointer variables.

The only limitation of reference variables is that they don't provide the flexibility of pointer variables. Once they are bound to a variable that binding cannot be changed. All accesses made to the reference variables are the same as the access to the variable. On the other side you can assign the address of any value variable of same type to a pointer variable of that type and change the binding of pointer variables at your will.

1.14 STREAM BASED I/O

The C++ I/O subsystem is designed to work with a wide variety of devices that includes keyboard, monitor, disks, tape drives, etc. The working principle (characteristics) of each of these devices is quite different. The C++ I/O subsystem is designed in such a way that it provides a uniform interface that is independent of the actual device being used.

Thus, the programmer can write a C++ program that can receive input from, and send output to any device without bothering about the characteristics of device being used. This interface is called a *stream*. Figure 1.1 shows this uniform interface.

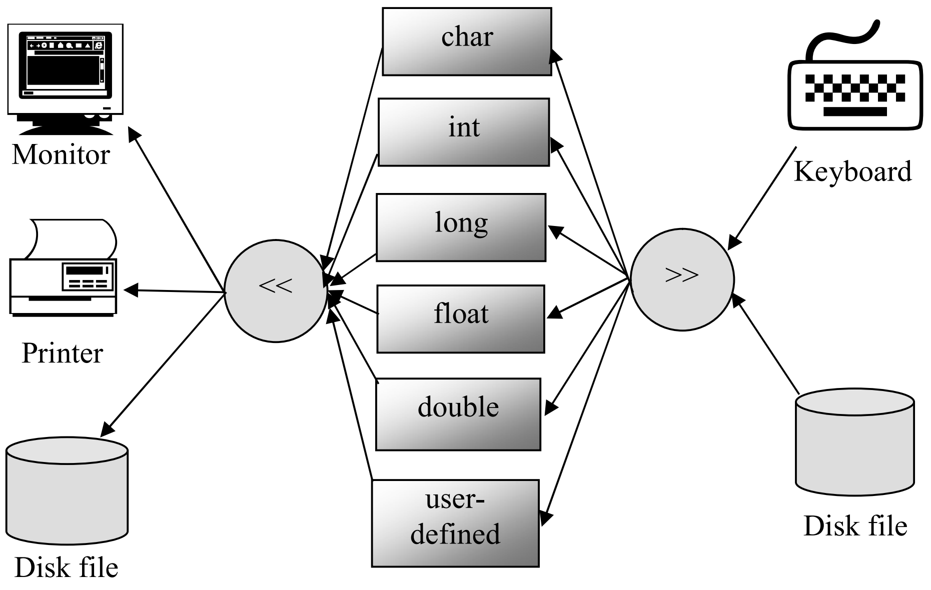
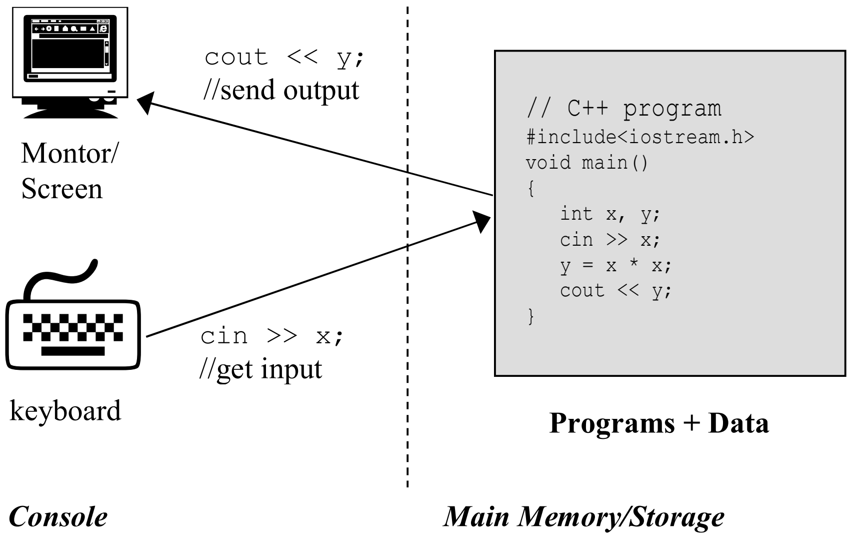
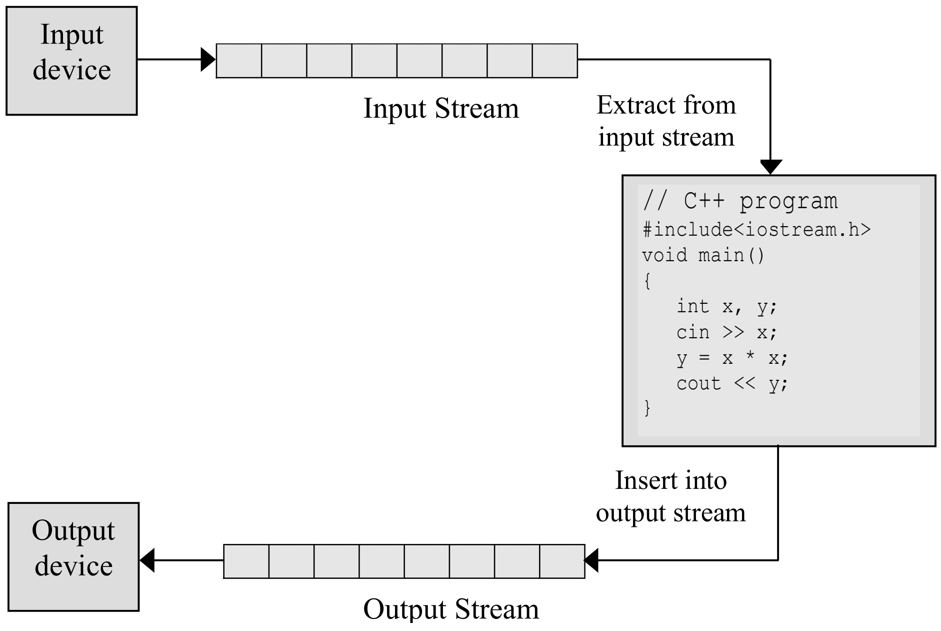


Figure 1.1: Consistent stream interface with I/O devices



```
// C++ program
#include<iostream.h>
void main()
{
   int x, y;
   cin >> x;
   y = x * x;
   cout << y;
}
```

Figure 1.2: Console-program interaction



```
// C++ program
#include<iostream.h>
void main()
{
   int x, y;
   cin >> x;
   y = x * x;
   cout << y;
}
```

Figure 1.3: Data streams and program interaction

Streams in C++ language are classified as

- o Input streams
- o Output streams
- o File streams

A stream basically is a sequence of bytes. It can either act as a *source* from which the program can take (extract) input data or as a *destination* to which the program can send (insert) output data. The source stream that supplies data to the program is called *input stream* and the destination stream that receives data from the program is called *output stream*. These streams collectively are called *data streams*.

Figure 1.3 shows the interaction of data streams with program.

1.15 INLINE FUNCTIONS

During program execution, a function call involves transfer of control to a specified address (address of first executable statement of the function), and returning to the instruction following the function call. Before transferring the control, CPU stores the contents of its registers and the address of the instruction following the function call, pushes actual arguments onto the system stack. After the transfer of control, the CPU pops the actual arguments from the system stack and copies into formal arguments that acts as local variables in the called function. Then it executes the function code, and stores the return value, if any, in a predefined memory location or register and returns control to the calling function. On return, the CPU retrieves the value returned by the function, if any, from the specified memory location or register, and resumes the executions from the instruction following the function call statement. The time taken during this whole process is called *context switch time*, and constitutes an overhead in the execution of the program.

This overhead is relatively large if the time required to execute a function is small than the context switch time. In that case, we would like to repeat the code in multiple places in a function rather than creating a separate function for it. The C++ language provides an alternative to above problem in the form of *inline functions*.

Inline functions are those functions whose body is inserted in place of the function call during the compilation process. Therefore, with inline functions, the program will not incur any context-switching overhead.

Hence, the inline functions enjoy both the flexibility and power of normal functions and macro functions. The inline functions are best used for small and frequently used functions.

An inline function definition is similar to an ordinary function except the keyword *inline* precedes the function definition.

This is illustrated in the following example:

```
inline float square( float x )
{
    return x*x;
}
int main()
{
    float a, b, r, z;
    // some statements
    a = square ( z );
    b = square ( r );
    // some statements
    return 0;
}
```

During the compilation process, the above code will be treated as

```
void main()
{
    float a, b, r, z;
    // some statements
    a = z * z;
    b = r * r;
    // some statements
}
```

1.15.1 Inline Functions Versus Macros (#define)

Given a choice, prefer *inline* over *#define.* Let us see why?

Like anything else in the world, there are advantages and disadvantages to everything and macros are not an exception to that. However, the list of advantages is over shadowed by list of disadvantages.

Consider the following macro:

```
#define MAX( a, b )  a < b ? b : a // 1
```

The problem here is that since variables *a* and *b* are not properly parenthesized, an expression in their place could produce side effects.

For example, macro call

```
MAX( x += 2, y )
```

expands to

```
x += 2 < y ? y : x += 2
```

which, by C++'s precedence rules, evaluates to

```
x += (( 2 < y ) ? y : x += 2 )
```

where as the required was

```
if ( x+ = 2 < y ) then result is y else result is x += 2
```

This pitfall can be taken care of in the following manner

```
#define MAX(a,b)  (a) < (b) ? (b) : (a)    // 2
```

Though the fix may look bug-free now, there is another similar subtler pitfall to it. The problem here is that though the single variables are parenthesised now, the whole expression is not properly parenthesised. Remember that a macro can be a part of larger expression and since it expands into plan stupid text, there is no evidence of a macro having been there once the expansion has taken place. If we do not want the parts of a macro to mix-and-match with others, we have to parenthesize the macro as a whole and not only its variables.

For example, the expression

```
z = MAX( a, b ) + 23
```

will expand to

```
z = ( a )< ( b ) ? ( b ):( a ) + 23
```

which, by C++'s precedence rules, evaluates to

```
z = (( a ) < ( b )) ? ( b ):(( a ) + 23 )
```

which is not the intended functionality.

The above problem can now be fixed by putting parentheses around the whole macro as

```
#define MAX( a, b )  (( a ) < ( b ) ? ( b ) : ( a )) // 3
```

This is not the end of problems that macro can create. Consider the use macro for the expression

```
MAX( ++x, y )
```

This would expand to

```
(( ++x ) < ( y ) ? ( y ):( ++x ))
```

which will twice increment the value of *x* in case ++*x* is greater than *y*. This is generally not the accepted result.

Another problem with macros is that about their scope. Macros always have global scope.

Some more disadvantages of macros are:

- *Macros do not have address* – Macros are not code. They do not have existence of their own in object code. They have been replaced by pre-processor in the resulting code.
- *Macros are debugger-unfriendly* – Till date, I think, there is no compiler that is macro friendly and provides similar functionality for stepping in/out, watching, etc. as it does for functions.
- *Macros are not type-safe* – Static type checking of arguments and return value is carried out by the compiler for functions, but not for macros.
- *Macros are not recursive* – Macros are not code. They do not have existence of their own in the object code. They have been replaced by the pre-processor.

The *inline* functions overcome all these disadvantages.

1.16 FUNCTION AS MEMBERS OF STRUCTURES

Structures in C language provide a mechanism to group elements (usually) of different data types in one unit that belong to the same family. For example, if we want to create a user-defined data type to represent an employee in an organization, the structure definition in C may look like

```
struct EMPLOYEE
{
    int empCode;
    char empName[20];
    float empSalary;
};
```

Here, for simplicity, we have considered only three elements viz. employee's code (*empCode*), employee's name (*empName*) and employee's salary (*empSalary*) while leaving out other elements such as date of birth, date of joining, educational and professional qualification, etc.

In order to process a variable (that represents an employee, an instance of family *employees*), either it must be declared as global variable so that it is accessible to all functions or must be passed as an argument to an appropriate set of functions in order to process it.

However, a structure in C++ LANGUAGE permits the inclusion of functions in addition to data members, and thus provides a true mechanism of handling data abstraction. The general syntax of structure in C++ LANGUAGE is

```
struct structureName
{
    public:
       // data and functions
    protected:
       // data and functions
    private:
       // data and functions
};
```

The structure has two types of members: *data member* and *function members*. Function defined in a structure can access any data member. The keywords *public*, *protected* and *private* are called *access specifiers*. If none of the above specifier appears in the structure definition, all the members have default access.

The private and protected members of a structure can only be accessed by the member function where as public members are accessible by members functions as well its instances (structure variables). Therefore, with appropriate use of these access specifiers, we can control the access to data members as desired in the application.

When we define a structure, member functions can be defined within the structure definition or outside the structure definition. When they are defined outside the structure definition, they are just declared (prototyped) in the structure definition and are bound to an appropriate structure definition by using scope resolution operator. Both of the ways are illustrated below:

When the member functions are defined inside the structure definition:

```
struct EMPLOYEE
{
   private:
      int empCode;           // private data members
      char empName[20];
      float empSalary;

   public:
      // public member function defined inside the class
      void readData()
      {
         cout << "Enter code : ";
         cin >> empCode;
         cout << "Enter name : ";
         cin >> empName;
         cout << "Enter salary : ";
         cin >> empSalary;
      }

      void showData()
      {
         cout << endl << "Employee information" << endl;
         cout << "Code   : " << empCode << endl;
         cout << "Name   : " << empName << endl;
         cout << "Salary : " << setprecision(2);
         cout << empSalary << endl;
      }
};
```

When the member functions are defined outside the structure definition:

```
struct EMPLOYEE
{
   private:
      int empCode;           // private data members
      char empName[20];
      float empSalary;
   public:
     // public member function declared inside the
     // class and defined outside the class
      void readData ();
      void showData ();
};
void employee :: readData()
{
   cout << "Enter code : ";
   cin >> empCode;
   cout << "Enter name : ";
   cin >> empName;
   cout << "Enter salary : ";
   cin >> empSalary;
}
void employee :: showData()
{
   cout << endl << "Employee information" << endl;
   cout << "Code   : " << empCode << endl;
   cout << "Name   : " << empName << endl;
   cout << "Salary : " << setprecision(2);
   cout << empSalary << endl;
}
```

The use of both of these definitions is illustrated in following programs.

Listing 1.7

```
/*
 * Program to illustrate the mechanism of defining
 * member functions inside the structure definition
*/
#include <iostream.h>
#include <iomanip.h>
struct EMPLOYEE
{
    private:
        int empCode;         // private data members
        char empName[20];
        float empSalary;
    public:
        // public member function defined inside the class
        void readData ()
        {
            cout << "Enter code : ";
            cin >> empCode;
            cout << "Enter name : ";
            cin >> empName;
            cout << "Enter salary : ";
            cin >> empSalary;
        }
        void showData ()
        {
            cout << endl << "Employee information" << endl;
            cout << "Code   : " << empCode << endl;
            cout << "Name   : " << empName << endl;
            cout << "Salary : " << setprecision(2);
            cout << empSalary << endl;
        }
};
int main()
{
    EMPLOYEE aEmployee;
    aEmployee.readData();    //member function invoked on instance
    aEmployee.showData();    //member function invoked on instance
    return 0;
}
```

Listing 1.8

```
/*
 * Program to illustrate the mechanism of defining
 * member functions outside the structure definition
 */
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
struct EMPLOYEE
{
     private:
        int empCode;         // private data members
        char empName[20];
        float empSalary;

     public:
        // public member function declared inside the
        // class and defined outside the class
        void readData ();
        void showData ();
};
void employee :: readData()
{
    cout << "Enter code : ";
    cin >> empCode;
    cout << "Enter name : ";
    cin >> empName;
    cout << "Enter salary : ";
    cin >> empSalary;
}
void employee :: showData() {
    cout << endl << "Employee information" << endl;
    cout << "Code   : " << empCode << endl;
    cout << "Name   : " << empName << endl;
    cout << "Salary : " << setprecision(2);
    cout << empSalary << endl;
}
int main()
{
    EMPLOYEE aEmployee;
    aEmployee.readData();    //member function invoked on instance
    aEmployee.showData();    //member function invoked on instance
    return 0;
}
```

The following points about structures in C++ language are important to remember:

1. When we compute the size of a structure, the member functions are not included in its size. This fact you can verify using *sizeof* operator. For example, for employee structure defines above, the *sizeof* operator will return value 26, *i.e.*, size of structure employee is 26 bytes (2 bytes for *empCode*, 20 bytes for *empName* and 4 bytes for *empSalary*).
2. The data member can be *private*, *protected* or *public*, but the member functions must be *public* otherwise it will not be possible to call member functions on a structure variable and thus serves no purpose.
3. If the access specifier for the data members is *private*, then a structure variable initialization cannot be combined with declaration. Therefore, we have to write a member functions to initialize a structure variable.

   Consider the following structure definition

```
struct DATE
{
    int day;  //data members with default access public
    int month;
    int year;
    // member functions
};
```

          or

```
struct DATE
{
    public:
       int day;     //public data members
       int month;
       int year;
      // member functions
};
```

Then compiler permits the following

```
DATE dd = { 10, 12, 2003 };
```

However, if the structure definition is

```
struct DATE
{
    private:
       int day;     //private data members
       int month;
       int year;
       // member functions
};
```

The compiler will not permit the following

```
DATE dd = { 10, 5, 2003 };
```

Therefore, in this case the only way is to use a member function as illustrated below:

```
struct DATE
{
    private:
       int day;     //private data members
       int month;
       int year;
       void initialize(int dd, int mm, int yy)
       {
           day = dd;
           month = mm;
           year = yy;
       }
       // more member functions
};
```

Now, we can declare a structure variable and then call member function *initialize()* to form the initialization.

```
DATE d;
d.initialize(10,5, 2003);
```

You may think that these two statements can be combined as we do in normal case, but it is not permitted. You first have to create a variable (instance), and only then you can call member function.

Therefore, following statement is not permitted:

```
DATE d.initialize(10,5, 2003);
```

Like structures, unions in C++ language also can have member functions.

1.17 DEFAULT ARGUMENTS

In C language, a function call must specify all the arguments used in the function definition. In a C++ function call, one or more arguments can be omitted with the condition that function is defined to take default values for the arguments that can be omitted by providing the default values in the function prototype or declarator.

In the function prototype or declarator, parameters without default arguments are placed first and those with default values are placed later. The arguments specified in the function call explicitly override the default values.

This following example illustrates the definition of a function with default arguments and how it can be used.

Listing 1.9

```
/*
 *  Program to illustrate the mechanism of defining
 *  function with default arguments
*/

#include <iostream.h>

void printline ( char ch = '-', int count = 80)
{
    cout << endl;
    for ( int i = 0; i < count; i++ )
```

```
        cout << ch;
    cout << endl;
}

int main()
{
    println();              // use both default arguments
    println('*');           // use second argument as default
    println('+', 45);       // use explicit arguments
    return 0;
}
```

The above program can also be written with function prototype as

Listing 1.10

```
/*
 *  Program to illustrate the mechanism of defining
 *  function with default arguments
*/

#include <iostream.h>

void printline ( char ch = '-', int count = 80); // prototype

int main()
{
    println();              // use both default arguments
    println('*');           // use second argument as default
    println('+', 45);       // use explicit arguments
    return 0;
}

void printline ( char ch, int count )
{
    cout << endl;
    for ( int i = 0; i < count; i++ )
        cout << ch;
    cout << endl;
}
```

The following prototype statement is also acceptable

```
void printline ( char = '-', int = 80 );
```

If you are using prototype for a function with default arguments, then the default values need to be specified only in the prototype. Specifying default values in the declarator too will lead to error indicating the default argument(s) is re-declared.

1.18 FUNCTION OVERLOADING

Suppose in our program we require the functionality to swap two values of i*nt* type, *float* type, *double* type, *char* type.

Normally, this situation is dealt with by writing four different functions with different names as shown below:

```
// function to swap two int values
void swapInt ( int &first, int &second ) {
   int temp;
   temp = first;
   first = second;
   second = temp;
}

// function to swap two float values
void swapFloat ( float &first, float &second ) {
   float temp;
   temp = first;
   first = second;
   second = temp;
}

// function to swap two double values
void swapDouble ( double &first, double &second ) {
   double temp;
   temp = first;
   first = second;
   second = temp;
}

// function to swap two char values
void swapChar ( char &first, char &second ) {
   char temp;
   temp = first;
   first = second;
   second = temp;
}
```

Thus we have four different functions with different names for identical functionality. It would have been nice if we can use single name instead of four different names.

In C++ language, it is possible to use the same function name to perform identical operation. Such functions are called *overloaded functions* and the process of defining such functions is called *function overloading*.

Overloaded functions must differ in argument list in either of the following ways:

- They may differ in number of arguments.
- They may differ in data types of arguments.
- They may differ both in number and type of arguments.

Therefore, in the above example case, we can use one name, say *swap*, for all four versions of the swap function to swap different values and depending on the type of actual arguments in the function call; the compiler will resolve the function call.

Listing 1.11

```
// Program to calculate the square of an int and float number
// using overloaded function sqr()
#include <iostream.h>
#include <iomanip.h>
int sqr ( int intArg );          // prototype for int version
float sqr ( float floatArg );    // prototype for float version
int main()
{
   int intNumber;
   float floatNumber;
   cout << "Enter integer number : ";
   cin >> intNumber;
   cout << "Enter float number : ";
   cin >> floatNumber;
   cout << endl << "square of " << intNumber << " = "
        << srq ( intNumber ) << endl;
   cout << "square of " << floatNumber << " = "
        << srq ( floatNumber ) << endl;
   return 0;
}
// function that returns square of a integer number
int sqr ( int intArg ) {
   return ( intArg * intArg );
}
// function that returns square of a float number
float sqr ( float intArg ) {
   return ( floatArg * floatArg );
}
```

Note that the return type does not play any role in function overloading. The reason is that it is not necessary to collect the returned value even if the called function returns a value.

This is demonstrated below:

Listing 1.12

```
// Program to demonstrate that return type doesn't play any
// role in overloading

#include <iostream.h>

void f(void);           // prototype
int f(void);            // prototype

int main()
{
   f();                 // function call
   return 0;
}
void f(void)            // function definition
{
   cout << "\nNo args., No return type\n";
}
int f(void)             // function definition
{
   cout << "\nNo args., int return type\n";
   return 10;
}
```

In function call, there will be ambiguity because just by seeing the call the system will not be able to figure out which version of function *f()* to call. Even, if we are not collecting the value, we cannot assume that function that does not return a value would be called.

Name Mangling – *Mechanism For Handling Function Overloading*

If there have been no overloading, then functions are always called by their name as given by the programmer. But, in case the compiler permits overloading, then to handle overloading, the compiler changes the names of all functions during compilations. This is known as *name mangling*.

Name mangling, also known as *name decoration*, is the process where the compiler changes the names of the functions to facilitate overloading.

1.19 FUNCTION TEMPLATES

The C++ language allows defining a single function capable of processing elements of different data types. Such a function is known as *function template* or *generic function*.

The syntax of function template is

```
template <class T1, class T2, . . .>
DataType FunctionName (arguments of type T1, T2, . . . )
{
   // local variables of type T1, T2, or any other type
   // function body, operating on variable of type T1, T2
   // any other variables
}
```

where *template* is the keyword for declaring function template

Following is the function template for swapping the values of two variables that are passed by reference.

```
Template <class T1>
void swap ( T1 &a, T1 &b )
{
    T1 temp;           // temporary variable
    temp = a;
    a = b;
    b = temp;
}
```

Following program illustrates the definition and use of function template.

Listing 1.13

```
/*
 * Program to illustrate mechanism of defining and
 * using function templates
*/

#include <iostream.h>

template <class T1>
void swap ( T1 &a, T1 &b)
{
    T1 temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    char ch1 = '*', ch2 = '+';
    int a1 = 10, a2 = 20;
    float b1 = 10.25, b2 = 22.75;
    double d1 = 12.67, d2 = 67.8;
    swap ( ch1, ch2 );
    cout << "\n" << ch1 << "," << ch2 << "\n";
    swap ( a1, a2 );
    cout << "\n" << a1 << "," << a2 << "\n";
    swap ( b1, b2 );
    cout << "\n" << b1 << "," << b2 << "\n";
    swap ( d1, d2 );
    cout << "\n" << d1 << "," << d2 << "\n";
    return 0;
}
```

1.20 DYNAMIC MEMORY MANAGEMENT OPERATORS

The C++ language provides the following operators to perform dynamic memory management:

- ***new*** **operator** – for dynamic memory allocation.
- ***delete*** **operator** – for dynamic memory deallocation.

1.20.1 The *new* Operator

The new operator allocates the memory in a manner similar to *malloc()* function in C language. The only difference is that it always return a pointer to an appropriate type, and there is no need for typecasting whereas the *malloc()* function returns a generic pointer (pointer to *void* data type) that must be typecasted to appropriate data type prior to its use otherwise system may behave inconsistently.

The *new* operator is defined as

```
type * new type [ size in integer ];
```

The following example illustrates its use.

```
int *intPtr;
intPtr = new int[100];
```

Allocate a memory block of 200 bytes, 2 bytes for one integer value, total of 100 integers.

Similar statements will be used for other primitive data types as well as user-defined data types.

```
struct DATE
{
    int day;
    int month;
    int year;
};
DATE *datePtr;
datePtr = new date;
```

Allocate memory for one structure item of type *DATE, i.e.,* 6 bytes.

The *new* operator also permits the initialization of memory locations during allocation.

The syntax for doing so is

```
type *ptrVar = new type ( InitialValue );
```

This is illustrated in following example:

```
int *intPtr = new int(100);
```

This statement allocates memory for an integer number and initializes it with value 100. The address of the memory allocated memory is assigned to pointer variable *intPtr*.

```
float *floatPtr = new float(125.75);
```

This statement allocates memory for a float number (single precision real number) and initializes it with value 125.75. The address of the memory allocated memory is assigned to pointer variable *floatPtr*.

1.20.2 The *delete* Operator

The *delete* operator is a counterpart of new operator and it deallocates (releases) memory allocated by the new operator back to the free poll of memory in a manner similar to *free()* function in C language.

The syntax of *delete* operator is defined as

```
delete pointerVariable;
```

The *free()* function in C language is defined as

```
void free ( void * pointerVariable );
```

The following statement in C++ language

```
delete ( ptrVar );
```

is equivalent to following statement in C language

```
free ( ptrVar );
```

where *ptrVar* is pointer variable that holds the address of the dynamically allocated memory using *new* operator and *malloc()* function, respectively. The memory allocated using *new* operator or *malloc()* function must be released by the *delete* operator and *free()* function, respectively.

Following program demonstrates the use of *new* and *delete* operators. The program reads two vectors of same size from keyboard, add these vectors and outputs the resultant vector.

Listing 1.14

```
/*
 *  Program to add two vectors. This program illustrates the use
 *  of new and delete operators
*/
#include <iostream.h>
#include <iomanip.h>
// function to input elements of a vector
void readVector ( float *vector, int size )
{
    for ( int i=0; i < size; i++ )
        cin >> vector[i];
}
// function to output elements of a vector
void writeVector ( float *vector, int size )
{
    for ( int i=0; i < size; i++ )
        cout << vector[i] << "  ";
    cout << endl;
}
// function to add to vectors
void addVectors ( float *x, float *y, float *z, int size )
{
    for ( int i=0; i < size; i++ )
        z[i] = x[i] + y[i];
}
int main()
{
    int vec_size;
    float *a, *b, *c;
    cout << "Enter size of vectors : ";
    cin >> vec_size;
    a = new float[vec_size];    // allocate memory for all vectors
    b = new float[vec_size];
    c = new float[vec_size];
    cout << "Enter elements of vector a : ";
    readVector(a, vec_size);
    cout << "Enter elements of vector b : ";
    readVector(b, vec_size);
    addVectors(a, b, c, vec_size);   // c = a + b
    cout << "Elements of resultant vector are : ";
    writeVector(c, vec_size);
    delete a;    // free memory allocated for all vectors
    delete b;
    delete c;
    // above three statements can also be combined as "delete a, b, c;"
    return 0;
}
```

QUICK RECAP . . .

In this chapter, we have learnt that

- C++ language is a superset of C.
- C++ language supports object-oriented programming in addition to procedural programming.
- C++ language supports new style of comments. Every comment line begins with two slash characters (//).
- C++ language supports two new data types – *bool* and *wchar_t*.
- For a structure, union and enumerated user-defined data types, you don't need to prefix keyword *struct*, *union* and *enum* before the *tag*.
- A variable in C++ language can be declared anywhere in the code.
- C++ language supports reference variables. A reference variable acts as alias for other variable.
- Reference variables enjoy the simplicity of value variables and power of pointer variables.
- A reference variable can only be bound to a value/pointer variable at the point of its declaration.

- A global variable can be accessed using *scope resolution operator* '**::**' (two consecutive colons).
- C++ language supports new style of typecasting that looks similar to function call.
- C++ language provides specialized cast operators – *static_cast*, *const_cast*, *dynamic_cast*, *reinterpret_cast*.
- C++ language provides operator keywords for *relational*, *bitwise* and some *short-hand* operators.
- New standard of C++ language also supports new style of header files, where you don't need to include the extension.
- New standard of C++ language also introduces the concept of namespaces that allows managing the large and complex software in an efficient manner.
- In addition to *call by value*, *call by address* (pointer), the arguments can also be passed using *call by reference* mechanism.
- C++ language introduces the concept of streams that streamlines the handling of input/output. Streams provide a uniform interface of doing the I/O irrespective of the I/O devices being used.
- C++ language also introduces the concept of *inline* functions. The inline functions are substitutes of *macros* that suffer from many inherit problems.
- In C++ language, structures can have function members in addition to data members.
- In C++ language, a function can have default arguments.
- C++ language supports the feature of function overloading, where you can give same function name for the functions that provide the same functionality but on different type of operands.
- C++ language also supports concept of *function templates* (*generic functions*), where you can define a single function capable of processing elements of different data types.
- C++ language provides two memory management operators – *new* and *delete*. These operators have added to support the requirements of object-oriented programming.

EXERCISE . . .

1. Trace the history of C++ language.
2. What are the types comments supported by C++ language?
3. Name the new basic data types added in C++ language over C language.
4. What is a reference variable?
5. Is it possible to access a global variable in a block when the local variable shares the name as that of a global variable? If yes, how?
6. Name the new cast operators in C++ language.
7. Name the operator keywords added in C++ language.
8. What is a namespace? How namespaces are useful?

9. By taking an example, demonstrate how the arguments can be passed using call by reference mechanism.
10. What is a stream? Name the predefined streams in C++ language.
11. What are inline functions? How they compare with normal functions?
12. What is function overloading? How is it useful?
13. What is function template? How is it useful?
14. Explain the usage of memory management operators incorporated in C++ language.

❑ ❑ ❑