ANNAI WOMEN'S COLLEGE
(Arts and Science)

Affiliated to Bharathidasan University-Tiruchirapalli.

TNPL Road,Punnamchathram,Karur-639136.

DEPARTMENT OF COMPUTER SCIENCE,COMPUTER APPLICATIONS & IT

Faculty Name: Mrs. R.BARANI,MCA,M.Phil.,(Ph.D).,NET.,

Major : I B.Sc(Computer Science)

Paper Code : 16SCCCS2

Title of Paper: Programming in C++

UNIT	TOPIC	PAGENO
1	Basic concepts of OOP	1-27
2	Classes and Objects	28-52
3	Inheritance	53-70
4	Managing Console I/O operations	71-102
5	Standard Template Library	103-111

CORE COURSE II

PROGRAMMING IN C++

Objective: To impart basic knowledge of Programming Skills in C++ language.

Unit I

Basic Concepts of Object- Oriented Programming - Benefits of OOP - Object Oriented Languages - Applications of OOP – Structure of C++ Program - Tokens, Expressions and Control Structures – Functions in C++

Unit II

Classes and Objects – Constructors and Destructors –Operator Overloading and Type Conversions

Unit III

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

Unit IV

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

Unit V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

Text Book

Balagursamy E, Object Oriented Programming with C++, Tata McGraw Hill Publications, Sixth Edition, 2013

Reference Books

Ashok Kamthane, Programming in C++, Pearson Education,2013.

UNIT-I
INTRODUCTION TO C++

Overview of C++:

- C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.
- C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.
- C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.
- C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Note − A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

Procedural Oriented Programming Vs Object Oriented Programming

	Procedural Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Basic Concepts of Object Oriented Programming

- It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:
    - **Objects**
    - **Classes**
    - **Data abstraction and encapsulation**
    - **Polymorphism**
    - **Inheritance**
    - **Dynamic binding**
    - **Message passing**

1) Objects:

- Object is a basic unit of OOPS.
- It has unique name.
- An object represents a particular instance of a class.
- We can create more than one objects of a class.
- The size of class is size of total number of data members of class.

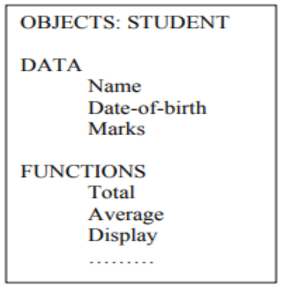


Fig. 1.5 representing an object

2) Classes:

- Class is the template of an object.
- That logically encapsulates data members and member functions into a single unit.
- Classes are data type based on which objects are created.

Eg:

Here we can take **Human Being** as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like Human Being, having body parts, and performing various actions.

3) Data Abstraction and Encapsulation:

- *Data abstraction* specifies hiding the implementation detail for simplicity. It increases the power of programming language by creating user define data types.
- *Data encapsulation* combines data members and member functions into a single unit that is called class. The advantage of encapsulation is that data cannot access directly. It is only accessible through the member functions of the class.

4) Polymorphism

- Polymorphism is basic and important concept of OOPS.
- Polymorphism specifies the ability to assume several forms.
- It allows routines to use variables of different types at different times.
- In C++, an operator or function can be given different meanings or functions.
- Polymorphism refers to a single function or multi-functioning operator performing in different ways.

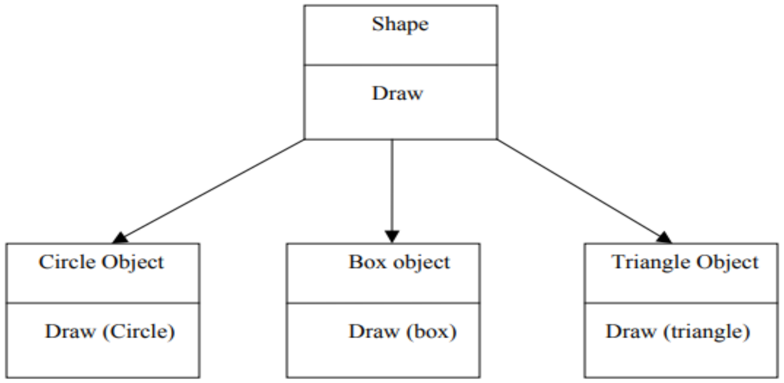
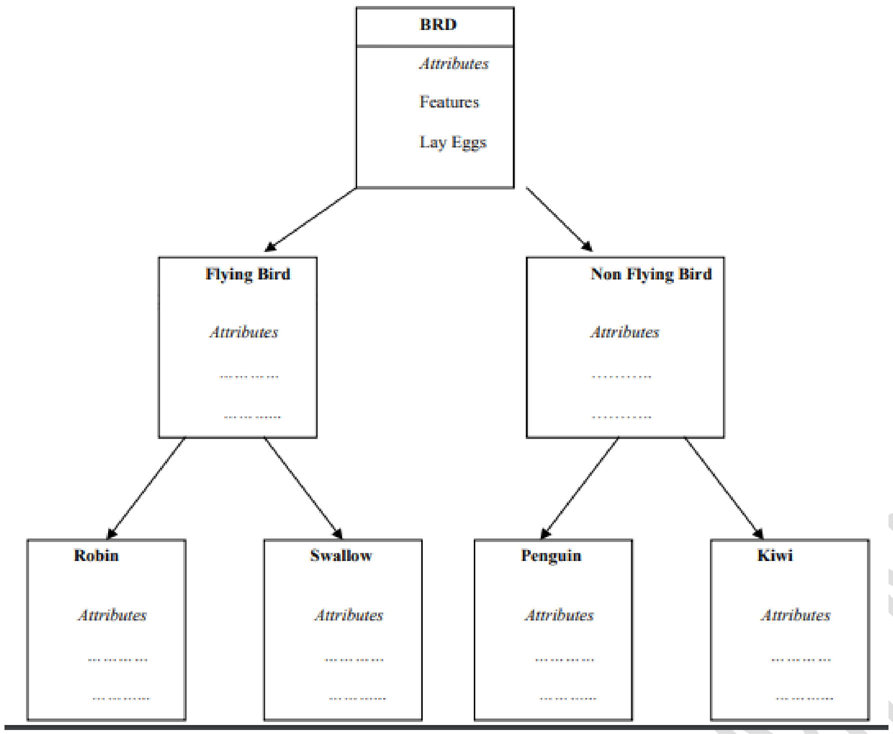


Fig: Polymorphism

5) Inheritance:

- Inheritance is the process of creating new class from existing class or base class.
- By using the concept of Inheritance, we can use implemented (existing) features of a class into another class.
- Base class is also known as parent class or super class.
- The new class that is formed is called derived class. It is also known as sub class or child class.
- Inheritance is basically used for reducing the overall code size of the program.

Fig. 1.6 Property inheritances

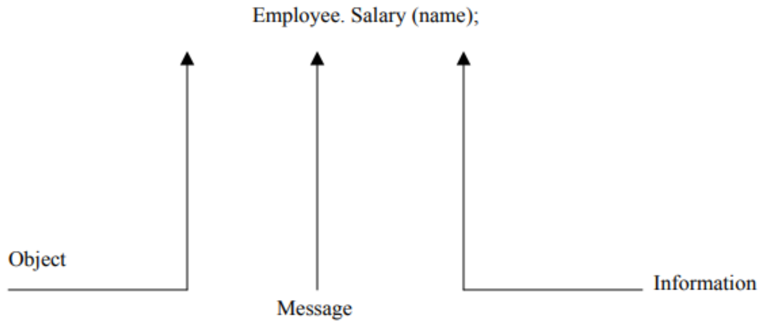


6) Dynamic Binding:

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance.
- A function call associated with a polymorphic reference depends on the dynamic type of that reference.

7) Message Passing:

- An object-oriented program consists of a set of objects that communicate with each other.
- The process of programming in an object-oriented language, involves the following basic steps:
  1. Creating classes that define object and their behavior,
  2. Creating objects from class definitions, and
  3. Establishing communication among objects.
- Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.
- A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.
- **Example:**



Object Oriented Language:

- Object-oriented programming is not the right of any particular languages. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal.
- However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially id designed to support the OOP concepts makes it easier to implement them.
- The languages should support several of the OOP concepts to claim that they are object-oriented.
- Depending upon the features they support, they can be classified into the following two categories:
  1. Object-based programming languages, and
  2. Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major feature that are required for object based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Object-oriented programming language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

Benefits of Object oriented Programming:

i. **Simplicity:** Software objects model real world objects, so the complexity is reduced and the program structure is very clear.
ii. **Modularity:** Each object forms a separate entity whose internal workings are decoupled from other parts of the system.
iii. **Modifiability:** It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
iv. **Extensibility:** adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
v. **Maintainability:** objects can be maintained separately, making locating and fixing problems easier.
vi. **Re-usability:** objects can be reused in different programs.

Applications of OOPS:

- Real-time system
- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

What is C++?

- "C++ is a statically-typed, free-form, (usually) compiled, multi-paradigm, intermediate-level general-purpose middle-level programming language."
- In simple terms, C++ is a sophisticated, efficient and a general-purpose programming language based on C. It was developed by Bjarne Stroustrup in 1979.
- Many of today's operating systems, system drivers, browsers and games use C++ as their core language. This makes C++ one of the most popular languages today.
- Since it is an enhanced/extended version of C programming language, C and C++ are often denoted together as C/C++.

Applications of C++:

- C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.
    - Since C++ allows us to create hierarchy-related objects, we can build special object-oriented libraries which can be used later by many programmers.
    - While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
    - C++ programs are easily maintainable and expandable.
    - It is expected that C++ will replace C as a general-purpose language in the near future.

Simple C++ Program

- Let us begin with a simple example of a C++ program that prints a string on the screen.

Printing A String

```
#include<iostream>
Using namespace std;
int main()
{
cout<<" c++ is better than c \n";
return 0;
}
```

Program 1.10.1

This simple program demonstrates several C++ features.

Program Features

- Like C, the C++ program is a collection of function.
- The above example contain only one function **main().**
- As usual execution begins at main(). Every C++ program must have a **main()**.
- C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces.
- Like C, the C++ statements terminate with semicolons.

Comments

- C++ introduces a new comment symbol // (double slash).
- Comment start with a double slash symbol and terminate at the end of the line.
- A comment may start anywhere in the line, and whatever follows till the end of the line is ignored.
- Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

// This is an example of
// C++ program to illustrate
// some of its features

The C comment symbols /*,*/ are still valid and are more suitable for multiline comments. The following comment is allowed:

/* This is an example of
C++ program to illustrate
some of its features
***/**

Output operator

- The only statement in program 1.10.1 is an output statement. The statement

  Cout << "C++ is better than C.";
- Causes the string in quotation marks to be displayed on the screen.
- This statement introduces two new C++ features**, cout** and **<<.**
- The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++.
- Here, the standard output stream represents the screen.
- It is also possible to redirect the output to other output devices.
- The operator << is called the insertion or put to operator.

The iostream File

- We have used the following #include directive in the program:

  #include <iostream>
- The #include directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file.
- The header file **iostream.h** should be included at the beginning of all programs that use input/output statements.

Namespace

- Namespace is a new concept introduced by the ANSI C++ standards committee.
- This defines a scope for the identifiers that are used in a program.
- For using the identifier defined in the **namespace** scope we must include the using directive, like

  Using namespace std;

Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. **Using** and **namespace** are the new keyword of C++.

Return Type of main()

- In C++, main () returns an integer value to the operating system.
- Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since main () returns an integer type for main () is explicitly specified as **int.**
- Note that the default return type for all function in C++ is **int.**
- The following main without type and return will run with a warning:

```
main ()
{
…………..
………….
}
```

More C++ Statements

Let us consider a slightly more complex C++ program. Assume that we should like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in program.

AVERAGE OF TWO NUMBERS

```
#include<iostream.h> // include header file
using namespace std;
int main()
{
float number1, number2,sum, average;
cin >> number1>> number2; // read numbers from keyboard
sum = number1 + number2;
average = sum/2;
cout << ”sum = “ << sum << “\n” << “average = “ << average << “\n”;
return 0;
} //end of example
```

The output would be:

```
enter two numbers: 6.5 7.5
sum = 14
average = 7
```

An Example with Class

- One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program shows the use of class in a C++ program.

USE OF CLASS

```
#include<iostream.h> // include header file
using namespace std;
class person
{
char name[30];
int age;
public:
void getdata(void);
void display(void);
};
void person :: getdata(void)
{
cout << “Enter name: “;
cin >> name;
cout << “Enter age: “;
cin >> age;
}
void person : : display(void)
{
cout << “\nNameame: “ << name;
cout << “\nAge: “ << age;
}
int main()
{
```

The output of program is:

```
Enter Name: Rithish
Enter age:20
Name:Rithish
Age: 20
```

```
person p;
p.getdata();
p.display();
return 0;
} //end of example
```

<u>Structure of C++ Program</u>

- ➢ As it can be seen from program 1.12.1, a typical C++ program would contain four sections as shown in fig. 1.9.
- ➢ This section may be placed in separate code files and then compiled independently or jointly.

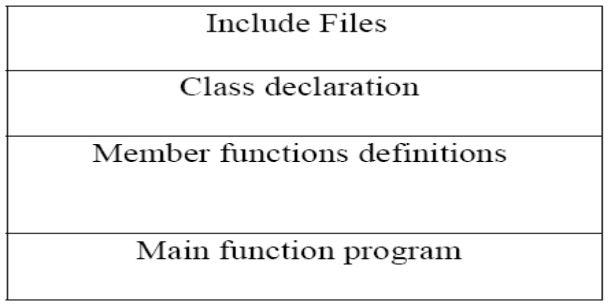
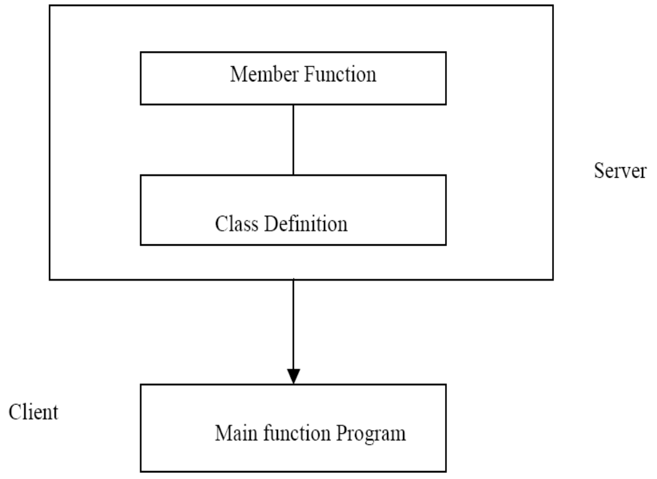


Fig 1.9 Structure of a C++ program

- ➢ It is a common practice to organize a program into three separate files.
- ➢ The class declarations are placed in a header file and the definitions of member functions go into another file.
- ➢ This approach enables the programmer to separate the abstract specification of the interface from the implementation details (member function definition).
- ➢ Finally, the main program that uses the class is places in a third file which "includes" the previous two files as well as any other file required.

Fig. 1.10 The client-server model



- ➢ This approach is based on the concept of client-server model as shown in fig. 1.10. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

<u>TOKENS:</u>

- A token is a group of characters.
- It is the smallest element of a C++ program which is meaningful to the compiler.

C++ uses the following types of Tokens:
1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators

<u>1. Keywords</u>

- Keywords are the reserved identifiers that have special meanings.
- These reserved keywords cannot be used as identifiers in a program.
- All keywords are written in lower case.

Following are the keywords used in C++:

asm	default	Float	Operator	static_cast	union
auto	delete	For	Private	struct	unsigned
break	do	Friend	Protected	switch	using
bool	double	Goto	Public	template	virtual
case	dynamic	If	Register	this	void
catch	else	Inline	reinterpret_cast	throw	volatile
char	enum	Int	Return	true	wchar_t
class	explicit	Long	Short	try	while
const	extern	Mutable	Signed	typedef	
const_cast	export	Namespace	Sizeof	typeid	
continue	false	New	Static	typename	

2. Identifiers

➢ Identifier is a sequence of characters used to define various things like variables, constants, functions, classes, objects, structures, unions etc.

It follows the rules for the formation of an identifier:

- An identifier consists of alphabets, digits or underscores.
- It cannot start with a digit. It can start either with an alphabet or underscore.
- Identifier should not be a reserved word.
- C++ is case-sensitive. So, upper case and lower case letters are considered different identifiers from each other.
- Blank spaces and special symbols are not allowed except underscore.

3. Constants

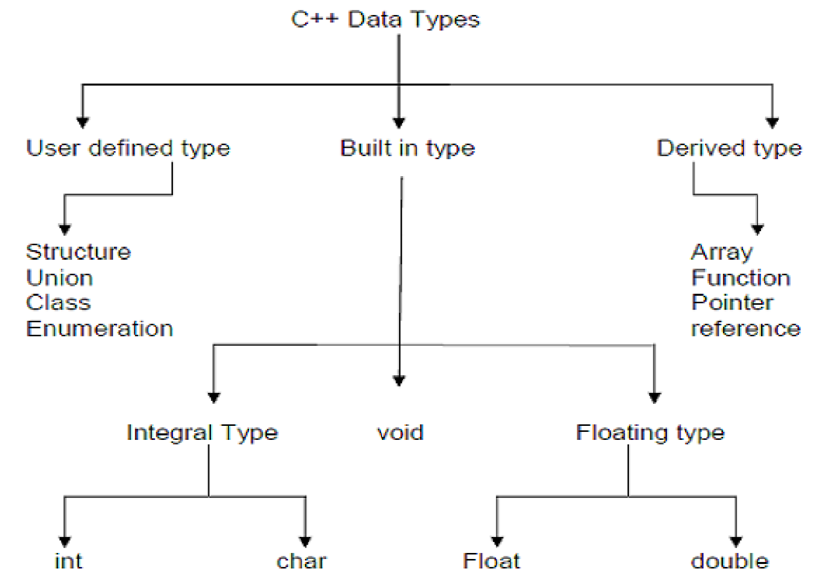
➢ Constants are normally the variables.
➢ The only thing that differentiates Constants from Variables is the fact that it is not allowed to modify the value of a constant by the program after the constants have already been defined.
➢ Constants refer to fixed values.
➢ Constants are also sometimes referred as **Literals.**
➢ They may belong to any of the data types.

Constant types

- **Integer constants:** For example: 0, 5, 957, 12376 etc.
- **Floating Point / Real constants:**
  For example: 0.7, 8.65, 4587.05 etc.
- **Octal and Hexadecimal constants:**
  For example:
  Octal: $(15)_8 = (13)_{10}$
  Hexadecimal: $(015)_{16} = (21)_{10}$
- **Character constants:** For example: ‘a’, ‘A’, ‘x’, ‘Z’ etc.
- **String constants:** For example: “Programming in C++”

BASIC DATA TYPES:

- Data types define the type of data a variable can hold, for example an integer variable can hold integer data, a character type variable can hold character data etc.
- Data types in C++ are categorised in three groups: **Built-in**, **user-defined** and **Derived**.



1) Integral type : – The data types in this type are int and char. The modifiers signed, unsigned, long & short may be applied to character & integer basic data type. The size of int is 2 bytes and char is 1 byte.

2) Void – Void is used:

i) To specify the return type of a function when it is not returning any value.

ii) To indicate an empty argument list to a function.

Ex- void function1(void)

iii)In the declaration of generic pointers.

Ex- void *gp

A generic pointer can be assigned a pointer value of any basic data type.

```
Ex. int *ip    // this is int pointer
gp = ip        //assign int pointer to void.
```

A void pointer cannot be directly assigned to their type pointers in c++ we need to use cast operator.

```
Ex – void *ptr1;
char *ptr2;
ptr2 = ptr1;   //   is allowed in c but not in c++.
ptr2 = (char *)ptr1;   // is the correct statement.
```

3)Floating type:

- The data types in this are float & double.
- The size of the float is 4 byte and double is 8 byte.
- The modifier long can be applied to double & the size of long double is 10 byte.

USER DEFINED DATA TYPES:

i) User-defined data type **structure and union** are same as that of C.

ii) **Classes** – Class is a user defined data type which can be used just like any other basic type once declared. The class variables are known as objects.

iii) **Enumeration**

a) An enumerated data type is another user defined type which provides a way of attaching names to numbers to increase simplicity of the code.

b) It uses enum keyword which automatically enumerates a list of words by assigning them values 0, 1, 2,…..etc.

Syntax:-

```
enum shape {   circle, square, triangle   };
enum color { black, blue, red   };
```

Now shape becomes a new type name & we can declare new variables of this type.

Ex . shape oval;

c) In C++, enumerated data type has its own separate type. Therefore c++ does not permit an int value to be automatically converted to an enum value.

```
Ex.   shape   shapes1  =  triangle;   //  is  allowed
      shape   shape1 = 2;                // Error in c++
      shape   shape1 = (shape)2;         //  ok
```

d) By default, enumerators are assigned integer values starting with 0, but we can override the default value by assigning some other value.

Ex.

enum colour {red, blue, pink = 3}; //it will assign red to 0, blue to 1, & pink to 3 or

enum colour {red = 5, blue, green}; //it will assign red to 5, blue to 6 & green to 7.

DERIVED DATA TYPES:

1) **Arrays:**

An array in c++ is similar to that in c, the only difference is the way character arrays are initialized. In c++, the size should be one larger than the number of character in the string where in c, it is exact same as the length of string constant.

```
Ex:   char string1[3]= “ab”; // in c++
      char string1[2] = “ab”; // in c.
```

2) **Functions**

Functions in c++ are different than in c there is lots of modification in functions in c++ due to object orientated concepts in c++.

3) **Pointers**: Pointers are declared & initialized as in c.

Ex: **int * ip; // int pointer**
ip = &x; //address of x through indirection

c++ adds the concept of constant pointer & pointer to a constant pointer.

char const *p2 = .HELLO.; // constant pointer

SYMBOLIC CONSTANT:

- Symbolic constant is a way of defining a variable constant whose value cannot be changed.
- It is done by using the keyword **const.**
- An identifier that represents a constant value throughout the life of the program is known as Symbolic Constants. It allows programmers to attach meaningful names to data values and hence enhances the readability of the programs.
- The named constants are just like variables except that their values cannot be changed. C++ requires a cont to be initialized but ANSI C does not require an initializer.
- If none is given, it initializes the const to 0. Constants are visible even outside the file in which they are declared. However , they can be made local by declaring them as static.
- To give a const value external linkage so that it can be referenced from another file, we must explicitly define it as an extern in C++.
- ***There are two ways of creating symbolic constants in C++ :-***
  1. Using the qualifier const.
  2. Defining a set of integer constants using enum keyword

  Syntax:

  const <Data_Type> <Variable_Name>;

  For example:

  const int c = 5;
- In C symbolic constant can be achieved by the use of **#define**.

  For example:

  #define PI = 3.142;

The general form of creating Symbolic constant Functions is :

<Return type> <Funciton_Name> ( ) const

Let us see an simple example of how to use const keyword with variables and functions :-

```
#include <iostream>
using namespace std;

const int a = 100;    // Const Variable

class TestConst
{
        public:
           void display() const   // Const Function
           {
                 cout << "Value of a in the const function is " << a;
            }
};

int main ( )
{
        Test int1;
        int1.display();
        cout << a;
        return 0;
            }
```

VARIABLE DECLARATION AND INITIALIZATION

- C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use.
- This informs the compiler the size to reserve in memory for the variable and how to interpret its value.
- The syntax to declare a new variable in C++ is straightforward: we simply write the type followed by the variable name (i.e., its identifier).

Syntax:

```
char c;     //character variable declaration.
int area;   //integer variable declaration.
float num;  //float variable declaration.
```

These are three valid declarations of variables.

- The first one declares a variable of type **char** with the identifier **c**.
- The second is declares a variable of type **int** with the identifier **area**.
- The third one declares a variable of type **float** with the identifier **num**.

Once declared, the variables **c**, **area** and **num** can be used within the rest of their scope in the program.

Declaring More than one Variable

- If declaring more than one variable of the same type, they can all be declared in a single statement by separating their identifiers with commas.

```
int a, b, c;    //more than one variable declaration.
```

This declares three variables (**a**, **b** and **c**), all of them of type **int**, and has exactly the same meaning as:

```
int a;  //integer variable declaration.
int b;  //integer variable declaration.
int c;  //integer variable declaration.
```

Initialization of variables

- When the variables in the example above are declared, they have an undetermined or garbage value until they are assigned a value for the first time.
- But it is possible for a variable to have a specific value from the moment it is declared. This is called the *initialization* of the variable.
- In C++, there are same ways to initialize variables as in C Language.

Syntax:

```
type identifier = initial_value;
```

Example:

```
int a = 10;  //integer variable declaration & initialization.
```

Practical:

```
//Write a CPP program for declaration & initialization of variable
#include <iostream.h>
int main ()
{
   int sum;    //Variable declaration
   int a = 10; //Variable declaration & initialization
   int b = 5;  //Variable declaration & initialization
   ans = a + b;
   cout << "Addition is:" << ans << endl;
   return 0;
}
```

Output:

```
        Addition is : 15
```

C++ REFERENCE VARIABLE:

- C++ introduces a new kind of variable known as ***Reference Variable.*** It provides an alias (alternative name) for a previously defined variable.
- A reference variable must be initialized at the time of declaration. This establishes the correspondences between the reference and the data object which it name.
- When a reference is created, you must tell it which variable it will become an alias for. After you create the reference, whenever you use the variable, you can just treat it as though it were a regular integer variable. But when you create it, you must initialize it with another variable, whose address it will keep around behind the scenes to allow you to use it to modify that variable.

Declaration:

[data_type] & [reference_variable]=[regular_variable];

regular_variable is a variable that has already initialized, and reference_variable is an alternative name (alias) to represent the variable regular_variable.

Consider the example

```
#include <iostream.h>
int main()
{
        int student_age=10;
        int &age=student_age;        // reference variable

        cout<< " value of student_age :"<< student_age << endl;
        cout<< " value of age :"<< age << endl;

        age=age+10;
        cout<<"\nAFTER ADDING 10 INTO REFERENCE
VARIABLE \n";
        cout<< " value of student_age :"<< student_age << endl;
        cout<< " value of age :"<< age << endl;
        return 0;
}
```

Output:

```
value of student age : 10
value of age : 10

AFTER ADDING 10 INTO REFERENCE VARIABLE
value of student age: 20
value of age : 20
```

OPERATORS IN C++

- C++ has a rich set of operators.
- All C operators are valid in C++ also. In addition, C++ introduces some new operators.
- We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >> .
- Other new operators are:

: :	Scope resolution operator
: :*	Pointer –to-member declarator
->*	Pointer – to- member operator
.*	Pointer – to- member operator
delete	Member release operator
endl	Line feed operator
new	Memory allocation operator
setw	Field width operator

- In addition, C++ also allows us to provide new definitions to some of the built-in operators.
- That is, we can give several meanings to an operator, depending upon the types of arguments used.
- This process is known as *operator overloading.*

SCOPE RESOLUTION OPERATOR

- C++ is also a Block-Structured Language.
- The Scope of a variable extends from the point of its Declaration till the end of the code block, containing the declarations.
- A Variable declared inside a code block is said to be local to that code block.
- **: :** (Scope Resolution Operator) Operator allows access to the global version of a Variable.

1) Global Scope Resolution Operator :
Let us see an program Example illustrating the use of Scope Resolution Operator (::) used with the Gloal Variable is given below :

```
#include <iostream.h>
 char c = 'a';     // global variable

void main() {
char c = 'b';   //local variable

cout << "Local  variable: " << c << "\n";
cout  <<  "Global  variable:  "  <<  ::c  <<  "\n";  //using  scope
resolution operator

}
```

Output:

```
Local variable: b
Global variable: a
```

2) Class Scope Resolution Operator :

In the below example we are using Scope Resolution Operator to define the class functions outside the class :

```
#include <iostream.h>
 class   programming
{
public:
void output ( );  //function declaration
};

// function definition outside the class
void programming::output( )
{
cout << "Function defined outside the class.\n";
}

int main( ) {
programming    x;
x.output( );
return 0;
}
```

Output:

```
Function defined outside the class.
```

MEMBER DEREFERENCING OPERATORS

- As you know, C++ permits us to define a class containing various types of data and functions as members.
- C++ also permits us to access the class members through pointers.
- In order to achieve this, C++ provides a set of three pointer-to-member operators.
- The below Table shows these operators and their functions.

Operator	**Function**
: : *	To declare a pointer to a member of a class.
*****	To access a member using object name and a pointer to that member.
->*	To access a member using a pointer to the object and a pointer to that member.

MEMORY MANAGEMENT OPERATORS

- **Arrays** can be used to store multiple homogenous data but there are serious drawbacks of using arrays.
- Programmer should allocate the memory of an array when they declare it but most of time, the exact memory needed cannot be determined until runtime.
- The best thing to do in this situation is to declare the array with maximum possible memory required (declare array with maximum possible size expected) but this wastes memory.
- So, To **avoid wastage of memory**, you can dynamically allocate the memory required during runtime using **new** and **delete** operator.

What are memory management operators?

There are two types of memory management operators in C++:

- **new**
- **delete**

These two memory management operators are used for allocating and freeing memory blocks in efficient and convenient ways.

New Operator:

- The new operator in C++ is used for dynamic storage allocation. This operator can be used to create object of any type.

  Syntax:

  ```
  pointer variable = new datatype;
  ```

- In the above statement, new is a keyword and the pointer variable is a variable of type data-type.

For example:

```
1. int  *a = new int

2. *a = 20;
      or
3. int  *a = new int(20);
```

- In the above example, the **new** operator allocates sufficient memory to hold the object of datatype int and returns a pointer to its starting point.
- the pointer variable a holds the address of memory space allocated.

Delete Operator:

- The **delete** operator in C++ is used for **releasing memory** space when the object is no longer needed.
- Once a new operator is used, it is efficient to use the corresponding delete operator for release of memory.

  Syntax:

  ```
  delete   pointer-variable;
  ```

For example:

```
#include <iostream.h>
 void main()
{
//Allocates using new operator memory space in memory for storing a integer datatype
int *a= new int;
*a=100;
cout << " The Output is:a= " << *a;
//Memory Released using delete operator
delete a;
}
```

Output:

```
The Output is  : a = 100
```

In the above program, the statement:

int *a= new a;

Holds memory space in memory for storing a integer datatype.

C++ MANIPULATORS

- Manipulators are operators used in C++ for **formatting output**. The data is manipulated by the programmer's choice of display.
- In this C++ tutorial, you will learn what a manipulator is, **endl** manipulator, **setw** manipulator, **setfill** manipulator and **setprecision** manipulator are all explained along with syntax and examples.

endl Manipulator:

- This manipulator has the same functionality as the 'n' newline character.

For example:

1. **cout** << "**Welcome**" << **endl;**
2. **cout** << " **Girls**";

setw Manipulator:

- This manipulator sets the minimum field width on output.

  Syntax:

  ```
  setw(x)
  ```

- Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator.
- The header file that must be included while using setw manipulator is .

- ***Example:***

```
#include <iostream>
#include <iomanip>
 void main( )
{
int     x1=123,x2=     234,
x3=789;
cout << setw(8) << "Exforsys" << setw(20) << "Values" << endl
<< setw(8) << "test123" << setw(20)<< x1 << endl
<< setw(8) << "exam234" << setw(20)<< x2 << endl
<< setw(8) << "result789" << setw(20)<< x3 << endl;
}
```

Output:

```
test        123
exam         234
result        789
```

setfill Manipulator:

- This is used after setw manipulator.
- If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

Example:

```
#include <iostream>
#include <iomanip>
void main( )
{
cout << setw(15) << setfill('*') << 99 << 97 << endl;
}
```

Output:

```
***********9997
```

setprecision Manipulator:

- The setprecision Manipulator is used with floating point numbers.
- It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

1. fixed
2. scientific

- These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator.
- The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation.
- The keyword scientific, before the setprecision manipulator, prints the floating point number in scientific notation.

Example:

```
#include <iostream,h>
#include <iomanip.h>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << scientific << x << endl;
}
```

Output:

```
c
o    0.100
u    1.000e-001
```

- The first cout statement contains fixed notation and the setprecision contains argument 3.
- This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation.
- The default value is used since no setprecision value is provided.

C++ TYPE CASTING

- A cast is a special operator that forces **one data type** to be **converted into another**.
- As an operator, a cast is unary and has the same precedence as any other unary operator.

Syntax:

```
type-name (expression)      //c++ notation
```

Example:

```
#include <iostream>

main()
{
double a = 99.09399;
float b = 97.20;
int c ;

c = (int) a;
cout << "Line 1 - Value of (int)a is :" << c << endl ;

c = (int) b;
cout << "Line 2 - Value of (int)b is  :" << c << endl ;
  return 0;
}
```

OUTPUT:

```
Line 1 - Value of  (int)a is :99
Line 2 – Value of (int)b is :97
```

EXPRESSIONS AND THEIR TYPES

- A combination of variables, constants and operators that represents a computation forms an expression.
- Depending upon the type of operands involved in an expression or the result obtained after evaluating expression, there are different categories of an expression.
- These categories of an expressions are:

Constant expressions: The expressions that comprise only constant values are called constant expressions. Some examples of constant expressions are 20, ‘ a‘ and 2/5+30 .

Integral expressions: The expressions that produce an integer value as output after performing all types of conversions are called **integral expressions.**

For example, x, 6*x-y and 10 + int (5.0) are integral expressions. Here, x and y are variables of type float.

Float expressions: The expressions that produce floating-point value as output after performing all types of conversions are called **float expressions.**

For example, 9.25, x-y and 9+ float (7) are float expressions. Here, x 'and yare variables of type float.

Relational or Boolean expressions: The expressions that produce a bool type value, that is, either true or false are called **relational or Boolean expressions.**

For example, x + y<100, m + n==a-b and a>=b + c .are relational expressions.

Logical expressions: The expressions that produce a bool type value after combining two or more relational expressions are called **logical expressions.**

For example, x==5 &&m==5 and y>x I I m<=n are logical expressions.

Bitwise expressions: The expressions which manipulate data at bit level are called **bitwise expressions.**

For example, a >> 4 and b<< 2 are bitwise expressions.

Pointer expressions: The expressions that give address values as output are called **pointer expressions.**

For example, &x, ptr and -ptr are pointer expressions. Here, x is a variable of any type and ptr is a pointer.

SPECIAL ASSIGNMENT EXPRESSIONS:

An expression can be categorized further depending upon the way the values are assigned to the variables.

Chained assignment: **Chained assignment** is an assignment expression in which the same value is assigned to more than one variable, using a single statement. For example, consider these statements.

a = (b=20); or a=b=20;

In these statements, value 20 is assigned to variable b and then to variable a. Note that variables cannot be initialized at the time of declaration using chained assignment.

For example, consider these statements.

int a=b=30; // illegal
int a=30, int b=30; //valid

__Embedded assignment:__ Embedded assignment is an assignment expression, which is enclosed within another assignment expression. **For example,** consider this statement

a=20+(b=30); //equivalent to b=30; a=20+30;

In this statement, the value 30 is assigned to variable b and then the result of (20+ 30) that is, 50 is assigned to variable a. Note that the expression (b=30) is an embedded assignment.

__Compound Assignment:__ Compound Assignment is an assignment expression, which uses a compound assignment operator that is a combination of the assignment operator with a binary arithmetic operator. For example, consider this statement.

a + =20; //equivalent to a=a+20;

In this statement, the operator += is a compound assignment operator, also known as short-hand assignment operator.

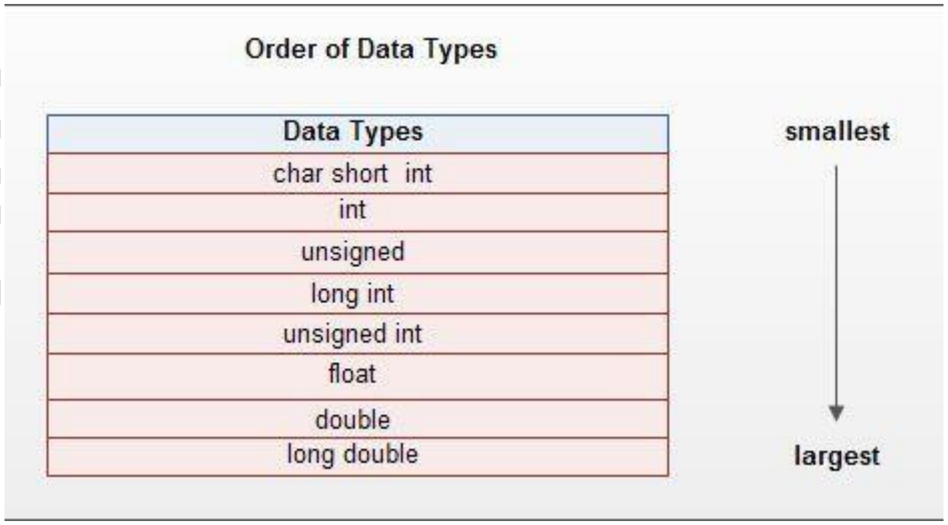
__TYPE CONVERSION__

An expression may involve variables and constants either of same data type or of different data types. However, when an expression consists of mixed data types then they are converted to the same type while evaluation, to avoid compatibility issues.

This is accomplished by type conversion, which is defined as the process of converting one predefined data type into another. Type conversions are of two types, namely***, implicit conversions*** **and** ***explicit conversions*** **also known as *typecasting.***

__Implicit Conversions__

- Implicit conversion, also known as automatic type conversion refers to the type conversion that is automatically performed by the compiler. Whenever compiler confronts a mixed type expression, first of all char and short int values are converted to int. This conversion is known as integral promotion.
- After applying this conversion, all the other operands are converted to the type of the largest operand and the result is of the type of the largest operand. Table illustrates the implicit conversion of data type starting from the smallest to largest data type.
- **For example,** in expression 5 + 4.25, the compiler converts the int into float as float is larger than int and then performs the addition.



Order of Data Types

Data Types
char short int
int
unsigned
long int
unsigned int
float
double
long double

smallest
largest

__Typecasting__

- Typecasting refers to the type conversion that is performed explicitly using type cast operator. In C++, typecasting can be performed by using two different forms which are given here.

data_type (expression) //expression in parentheses
(data_type)expression //data type in parentheses

where,

data_type = data type (also known as *cast operator)* to which the expression is to be converted.

To understand typecasting, consider this example.

float (num)+ 3.5; //num is of int type

In this example, float () acts as a conversion function which converts int to float. However, this form of conversion cannot be used in some situations. For example, consider this statement.

ptr=int * (x) ***;***

In such cases, conversion can be done using the second form of typecasting (which is basically C-style typecasting) as shown here.

ptr=(int*)x;

OPERATORS PRECEDENCE IN C++

- Operator precedence determines the grouping of terms in an expression.
- The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.
- This affects how an expression is evaluated.
- Certain operators have higher precedence than others;
- **For Example**, the multiplication operator has higher precedence than the addition operator**: x = 7 + 3 * 2;** here, **x** is assigned **13**, not **20** because operator ***** has higher precedence than **+,** so it first gets multiplied with **3*2** and then adds into **7.**

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

CONTROL STRUCTURES IN C++

- In C++, a large number of function& are used that pass messages, and process the data contained in objects.
- A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal.
- The format should be such that it is easy to trace the flow of execution of statements.
- This would help not only in debugging but also in the review and maintenance of the program later.

All programs use control structures to implement the program logic.

There are three types of Control Structures

1. **Sequence Structure (Straight line)**
2. **Selection Structure (Branching)**
3. **Loop Structure (Iteration or Repetition)**

Figure 3.4 shows how these structures arc implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

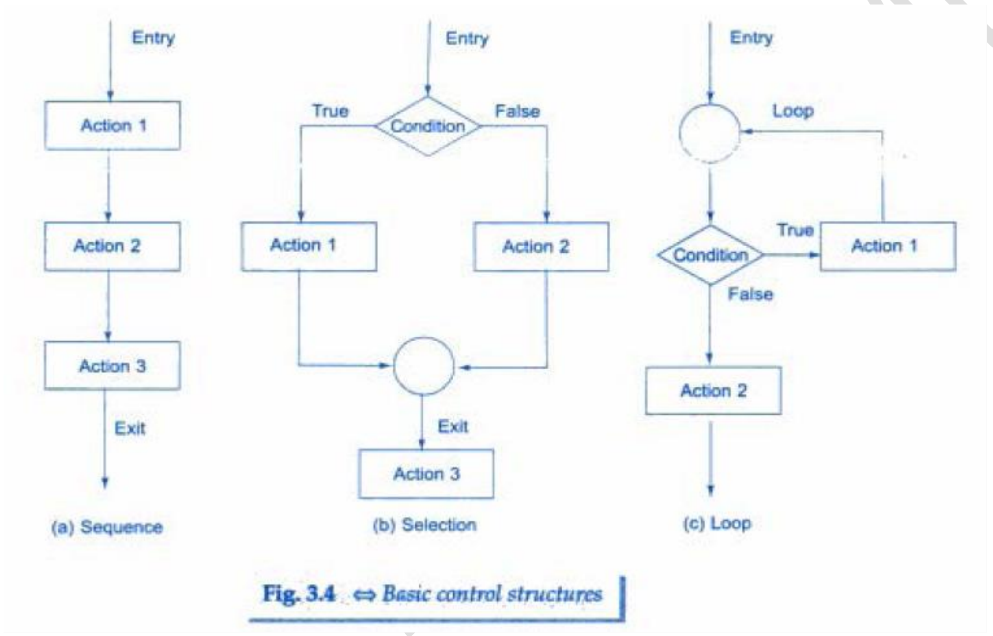


Fig. 3.4 ⇔ *Basic control structures*

- It is important to understand that all program processing can be coded by using only **these three logic structures.**
- The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique **in software engineering.**
- Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig. 3.6.
- This shows that C++ combines the power of structured programming with the object-oriented paradigm.

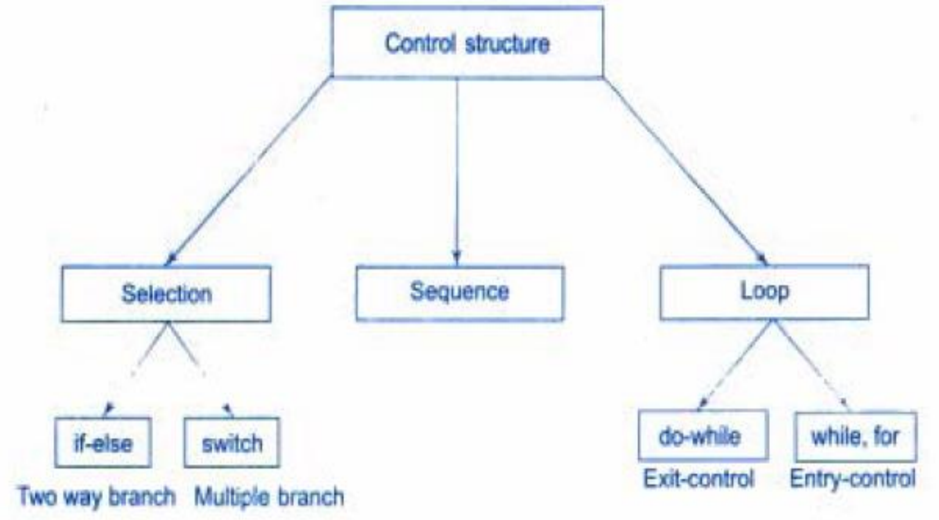


Fig. 3.6 ⇔ C++ *statements to implement in two forms*

The if statement

- The **if** statement is a powerful decision making statement and is used to control the flow of execution of statements.

  Syntax:

```
if( test expression)
```

The if statement is implemented in two forms:

• Simple if statement

• if ••• else statement

Simple If Statement :

✓ The general form of simple if statement is

```
if(test expression)
 {
   statement-block;
 }
   statement-x;
```

✓ The 'statement-block' may be a single statement or a group of statement.

✓ If the test expression is true the statement block will be executed.

✓ Otherwise the statement -block will be skipped and the execution will jump to the statement –x.

✓ If the condition is true both the statement–block and the statement -x are executed in the sequence .

If –Else Statement :

✓ The If statement is an extension of the simple if statement.

Syntax:

```
if (test expression)
{
true-block statements;
}
else
{
false-block statements;
}
statement – x;
```

- If the test expression is true then true-block statement are executed, otherwise the false –block statement are executed.
- In both cases either true-block or false-block will be executed not both.

Switch Statement :

- This is a multiple branching statement where, based on a condition, the control is transferred to one of the many possible points.

Syntax:

```
 switch (expression)
{
case value1   :
                    block1;
                    break;
case value 2  :
                    block 2;
                     break;
default       :
                   default block;
                    break;
   ……….
   ……….
}
statement – x;
```

The Do- While Statement:

- It is an exit – controlled loop.
- Based on a condition, the control is transferred back to a particular point in the program.

Syntax:

```
do
{
body of the loop;
}
while (test condition);
```

The While Statement:

- This is also a loop structure, but it is an entry-controlled one.
- In this the test condition is placed before the body of the loop.

Syntax:

```
while (test condition)
{
body of the loop;
}
```

The For Statement:

- It is an entry-controlled loop and is used when an action is to be repeated for a predetermined number of times.
- **Syntax:**

```
for (initialization;  test – condition ;  increment or decrement)
{
body of the loop;
}
```

FUNCTIONS

- **C++ functions** are a group of statements in a single logical unit to perform some specific task.
- Along with the main function, a program can have multiple functions that are designed for different operation.
- The results of functions can be used throughout the program without concern about the process and the mechanism of the function.

++ Functions

- In POP (Procedural Oriented Programming) language like C, programs are divided into different **functions** but in OOP (Object Oriented Programming) approach program is divided into objects where functions are the components of the object.

Generally, C++ function has three parts:

1. **Function Prototype**
2. **Function Definition**
3. **Function Call**

C++ Function Prototype

- While writing a program, we can't use a function without specifying its type or without telling the compiler about it.
- So before calling a function, we must either declare or define a function.
- Thus, declaring a function before calling a function is called **function declaration or prototype** which tells the compiler that at some point of the program we will use the function of the name specified in the prototype.

Syntax

```
return_type function_name (parameter_list);
```

Note: function prototype must end with a semicolon.

✓ Here, **return_type** is the type of value that the function will return. It can be int, float or any user-defined data type.

✓ **function_name** means any name that we give to the function. However, it must not resemble any standard keyword of C++.

✓ Finally, **parameter_list** contains a total number of arguments that need to be passed to the function.

C++ Function Call

- **Function call** means calling the function with a statement. When the program encounters the function call statement the specific function is invoked.

Syntax

```
function_name (argument_list );
```

✓ Here, **function_name** is the name of the called function and **argument_list** is the comma-separated list of expressions that constitute the arguments.

✓ The syntax is similar to that of prototype except that *return_type is not used.*

C++ Functions Definition

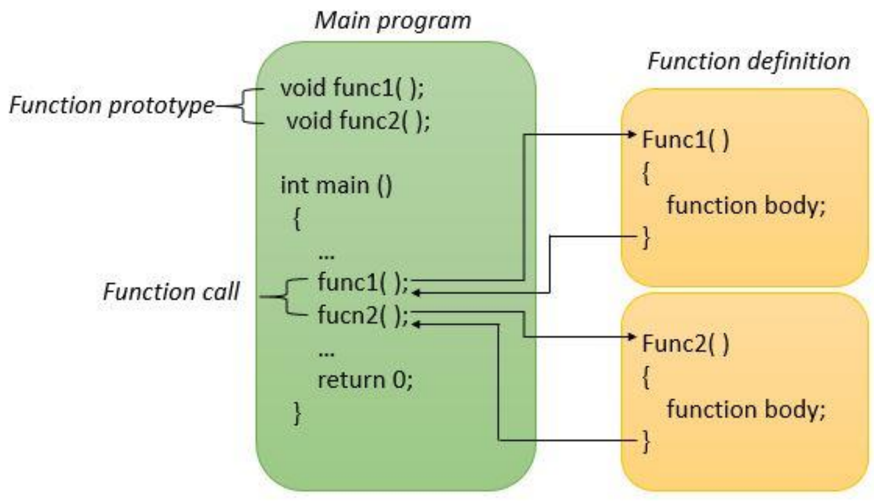
- **Function definition** is a part where we define the operation of a function. It consists of the declarator followed by the function body.

Syntax

```
return_type function_name( parameter_list )
{
   function body;
}
```

- ✓ Defining a function is a way of specifying how the operation is to be done when the function is called.

Illustration of function call



- ✓ Whenever we declare a function a part of the memory is reserved for that function and when we define the function is stored in that memory block.
- ✓ Let's suppose the function is stored at the address 0x111. When we call the function in the main program compiler goes to that memory location 0x111 where the code is executed as defined.
- ✓ After the execution, the compiler returns the result to the program without concerning any details how the result was obtained.
- ✓ Though it takes time for execution, it becomes handy when dealing with huge programs.

General structure of a function in C++ program

```
//Structure of C++ program

#include <iostream.h>
return_type function_name(parameter_list);    //function prototype

void main()
{
  ........
  function_name();   //function call
  ........
}

return_type function_name(parameter_list)     //function defintion
{
  ........
  function definition
  ........
}
```

CALL BY REFERENCE

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
- To pass the value by reference, argument reference is passed to the functions just like any other value.
- So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
  // function definition to swap the values.
  void swap(int &x, int &y)
{
   int temp;
   temp = x;     /* save the value at address x */
   x = y;        /* put y into x */
   y = temp;     /* put x into y */
     return;
}
```

For now, let us call the function **swap( )** by passing values by reference as in the following example −

```
#include <iostream.h>

void swap(int &x, int &y);    // function declaration

void main ( )
 {
   int a = 100 , b = 200;      // local variable declaration:

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   /* calling a function to swap the values using variable reference.*/
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

RETURN BY REFERENCE:

- We have studied the reference variable and it's functioning.
- A reference allows creating alias for the pre-existing variable.
- A reference can also be returned by the function.
- A function that returns reference variable is in fact an alias for referred variable.
- This technique of returning reference is used to establish cascade of member functions calls in operator overloading.

7.7 Write a program to return a value by reference.

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

int main()
{
   clrscr();
   int & min ( int &j, int &k);
   int a=18,b=11,c;
   c=min (a,b);
   cout<<"Minimum Value = "<<c;
   return 0;
}
int & min (int &j, int &k)
{
   if (k<j ) return k;
   else
   return j;
}
```

OUTPUT

```
Minimum Value = 11
```

Explanation: In the above program, the statement int & min (int &j, int &k) declares prototype of function min(). The '&' reference operator is used because the function returns reference to int and also receives arguments as reference. The function min() receives two integers as reference and returns minimum value out of two by reference.

INLINE FUNCTIONS

- C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.
- To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.
- A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers –

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

inline int max (int x, int y)
{
return (x > y)  ?  x : y;
}
int main( )      // Main function for the program
{
cout << " max (20,10 ) : "<< max(20,10) << endl;
cout << " max (0,200 ) : "<< max(0,200) << endl;
cout << " max (100,1010 ) : "<< max(100,1010) << endl;
return 0;
}
```

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

```
max (20,10 ) : 20
max (0,200 ) : 200
max (100,1010 ) : 1010
```

DEFAULT ARGUMENT

- A default argument is a function argument that has a default value provided to it. If the user does not supply a value for this argument, the default value will be used. If the user does supply a value for the default argument, the user-supplied value is used.

Note : Only the trailing arguments can have default values and therefore we must add default values form *right-to-left.*

Some examples of function declaration with default values are:

```
int Add(int x, int y, int z=30);              //Valid
int Add(int x, int y=20, int z=30);           //Valid
int Add(int x=10, int y=20, int z=30);        //Valid

int Add(int x=10, int y, int z);              //Invalid
int Add(int x=10, int y, int z=30);           //Invalid
```

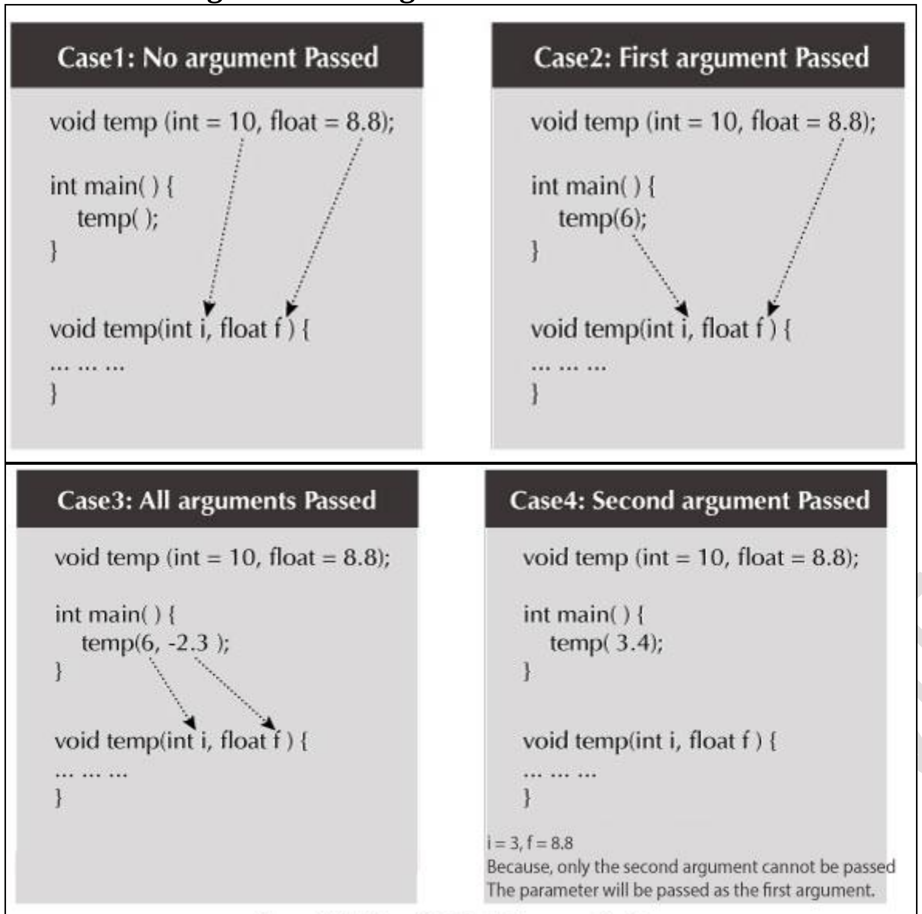
Example :

```
#include<iostream.h>
int Add(int x, int y=20, int z=30)
{
          return x + y + z;
}
void main( )
{
          int rs;

          rs = Add(5);
          cout<<"\n\tThe sum is : "<<rs;

          rs = Add(4,8);
          cout<<"\n\tThe sum is : "<<rs;

          rs = Add(7,3,4);
          cout<<"\n\tThe sum is : "<<rs;
}
```

Output :

```
The sum is : 55
The sum is : 42
The sum is : 14
```

Working of default arguments



```
Case1: No argument Passed

void temp (int = 10, float = 8.8);

int main( ) {
   temp( );
}

void temp(int i, float f ) {
... ... ...
}
```

```
Case2: First argument Passed

void temp (int = 10, float = 8.8);

int main( ) {
   temp(6);
}

void temp(int i, float f ) {
... ... ...
}
```

```
Case3: All arguments Passed

void temp (int = 10, float = 8.8);

int main( ) {
   temp(6, -2.3 );
}

void temp(int i, float f ) {
... ... ...
}
```

```
Case4: Second argument Passed

void temp (int = 10, float = 8.8);

int main( ) {
   temp( 3.4);
}

void temp(int i, float f ) {
... ... ...
}
```

i = 3, f = 8.8
Because, only the second argument cannot be passed
The parameter will be passed as the first argument.

Figure: Working of Default Argument in C++

<u>CONST ARGUMENTS</u>

- The constant variable can be declared using const keyword.
- The const keyword makes variable value stable.
- The constant variable should be initialized while declaring.

Syntax:

a) const < variable name > = <value>
b) <function name > (const <type> * <variable name>;)
c) int const x // invalid
d) int const x =5 // valid

In statement (a) , the const modifier enables to assign an initial value to a variable that cannot be changed later by the program.

Example:

const age = 40;

Any attempt to change the contents of const variable age will produce a compiler error. Using pointer one can indirectly modify a const variable as per the following:

*(int *) &age = 45;

Write a program to declare constant variable:

```
#include<iostream.h>
#include<conio.h>
int main()
{
   clrscr();
   int min (int const a=8, int b=20);
   int a=12,b=45;
   b=min (a);
   cout<<"\n a= "<<a <<" b= "<<b;
   return 0;
}
int min (int const j, int k)
{
   // j++; // can not modify a constant variable
   // k++; // valid because b is not constant
   cout<<"\n j= "<<j <<" k ="<<k;
   if (k<j ) return k;
   else return j;
}
```

OUTPUT

```
j= 12 k =20
a= 12 b= 12
```

RECURSION

- When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.
- A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Simple example of recursion.

```
recursionfunction( )
{
        recursionfunction();    //calling self function
}
```

Example to print factorial number using recursion :

```
#include<iostream.h>
void  main( )
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
}

int factorial(int n)
{
if(n<0)
return(-1);      /*Wrong value*/
if(n==0)
return(1);       /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Output:

```
Enter any number: 5
Factorial of a number is : 120
```

FUNCTION OVERLOADING

- Two or more functions having same name but different argument(s) are known as overloaded functions.
- Function refers to a segment that groups code to perform a specific task.
- In C++ programming, two functions can have same name if number and/or type of arguments passed are different.
- These functions having different number or type (or both) of parameters are known as overloaded functions.

For example:

```
int   test( ) {  }
int   test(int a) { }
float  test(double a) { }
int   test(int a, double b) { }
```

✓ Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

✓ Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

// Error code

```
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example : Function Overloading

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

void display(int);
void display(float);
void display(int, float);

void  main( )
{

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);

}

void display(int var) {
   cout << "Integer number: " << var << endl;
}

void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output:

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

- ✓ Here, the display() function is called three times with different type or number of arguments.
- ✓ The return type of all these functions are same but it's not necessary.

UNIT-I COMPLETED

UNIT-II
CLASSES AND OBJECTS

CLASSES:

- A class is a way to bind data and functions together in a single data type. The variables and functions enclosed in a class are called ***data members*** and ***member functions***. Since classes by default are private, class allows the data (and functions) to be hidden if necessary from external use.
- This mechanism of binding data and functions that operate on that data is call ***data encapsulation***.
- This mechanism of hiding data of a class from the outside world (other classes) so that any access to it either intentionally or unintentionally can't modify the data is called ***data hiding.***

Class Declaration:

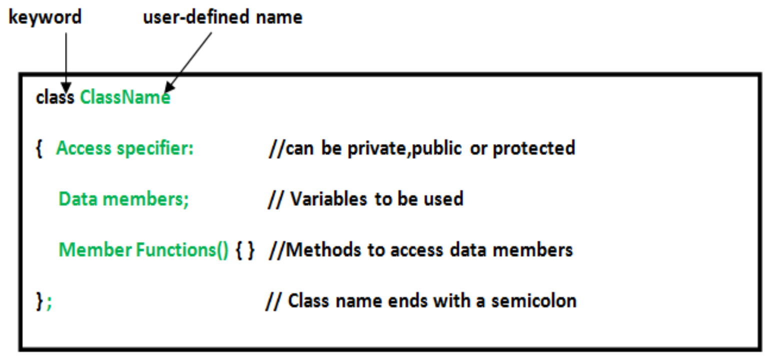
A class specification has two parts:

1. Class Declaration
2. Class Function definitions

The ***class declaration*** describes the type and scope of its member. The ***class definitions*** describe how the class functions are implemented.

The syntax of a class definition is shown below :

```
class name_of _class
 {
  private:   Variable declaration; // data member
             Function declaration; // Member Function

   protected: Variable declaration;
               Function declaration;

   public:     Variable declaration;
             Function declaration;
};
```



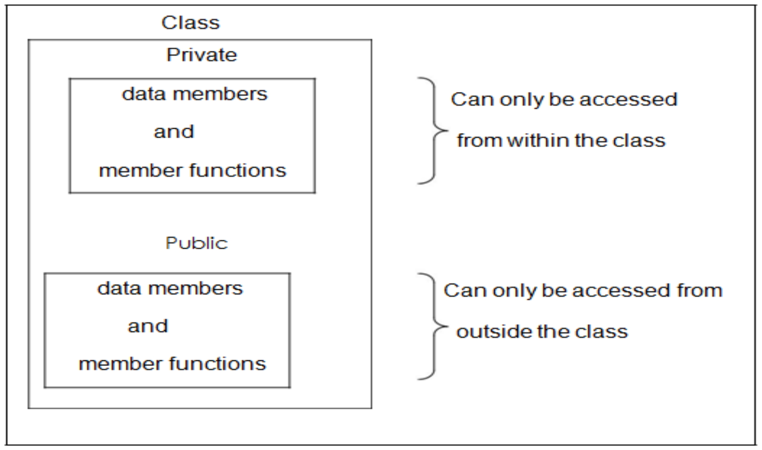
```
class ClassName
{  Access specifier:          //can be private,public or protected
   Data members;              // Variables to be used
   Member Functions() { }     //Methods to access data members
};                            // Class name ends with a semicolon
```

Here, the keyword class specifies that we are using a new data type and is followed by the class name.

Here, access-specifier is one of these three C++ keywords:

1. **public**
2. **private**
3. **protected**

By default, functions and data declared within a class are *private*. *Private* data and functions can only be accessed from within the class itself. *Public* data and functions are accessible outside the class also. The *protected access_specifier* is needed only when inheritance is involved.



Example:

```
        class student
        {
                int   rollno;
                float  marks;

                public:
                        void getdata ( );
                        void display ( );
        };
```

Creating Object:

- An object is an *instance* of a class. In general a class is a user defined data type, while an object is an instance of an class.

e.g. Let *student* be the name of class

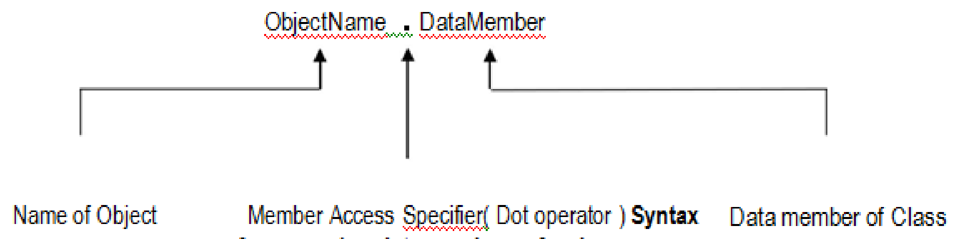
student S1, S2;

creates variables S1, S2 of type student. Once the objects are declared memory is allocated. Objects can also be created by placing their names immediately after the closing brace.

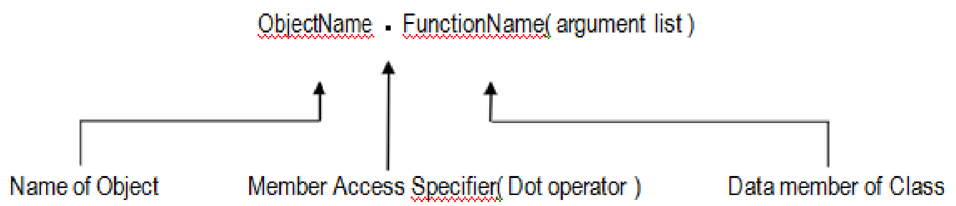
e.g. **class *student***
{
……
……
} S1, S2, S3;

Accessing Class members

- After creating the object there is a need to access the class member.
- This can be done by using a dot (**.**) operator.



Syntax for accessing data members of a class


ObjectName . FunctionName(argument list)
Name of Object
Member Access Specifier(Dot operator)
Data member of Class

Syntax for accessing member Function of a class

Example: Function call statement

S1.getdata(129,704.5); //assign value 129 to rollno and 704.5 to marks of object S1.
S1.display( ); //will display value of data members.

Similarly the statement
S1.roll = 129; is invalid because data member is private.

DEFINING MEMBER FUNCTIONS

- Member functions of a class can be defined either outside the class definition or inside the class definition.
- In both the cases, the function body remains the same, however, the function header is different.
- In C++, the member functions can be coded in two ways :
  (a) Inside class definition
  (b) Outside class definition using scope resolution operator **(::)**

Outside the Class:

- Defining a member function outside a class requires the function declaration (function prototype) to be provided inside the class definition. The member function is declared inside the class like a normal function.
- This declaration informs the compiler that the function is a member of the class and that it has been defined outside the class. After a member function is declared inside the class, it must be defined (outside the class) in the program.
- The definition of member function outside the class differs from normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator **(: :).**
- The scope resolution operator informs the compiler what class the member belongs to.

Syntax:

```
Return_type class_name :: function_name (parameter_list)
{
// body of the member function
}
```

Example :

```
class book
 {
   body of the class
 } ;
void book :: getdata(char a[ ],float b)
{
// defining member function outside the class
strcpy(title, a):
price = b:
}
void book :: putdata ( )
{
cout<<"\n Title of Book: "<<title;
cout<<"\n Price of Book: "<<price;
}
```

Inside the Class:

- A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

Example :

```
class book
{
char title[30];
float price;
public:
void getdata(char [ ],float);      // declaration
void putdata( )        //definition inside the class
{
cout<<"\n Title of Book: "<<title;
cout<<"\n Price of Book: "<<price;
} ;
```

In this example, the member function putdata( ) is defined inside the class book.
Hence, putdata( ) is by default an inline function.
Note that the functions defined outside the class can be explicitly made inline by prefixing the keyword inline before the return type of the function in the function header.

For example, consider the definition of the function getdata( ).

```
inline void book :: getdata (char a [ ], float b)
{
body of the function
}
```

A SIMPLE C++ PROGRAM WITH CLASS

PROGRAM 1:

```
#include< iostream.h>
#include<conio.h>
class hello;
{
public:
    void sayhello( )
          {
          cout<< “hello world”<< endl;
          }
};
void main( )
{
hello h;
h.sayhello( );
}
```

PROGRAM 2:

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

class student
{
    private:
          char name[20],  regd[10],  branch[10];
          int sem;
   public:
          void input( );
          void display( );
};

void student :: input( )
{
          cout<<"Enter Name:";
          cin>>name;
          cout<<"Enter Regd no.:";
          cin>>regd;
          cout<<"Enter Branch:";
          cin>>branch;
          cout<<"Enter Sem:";
          cin>>sem;
}

void student :: display( )
{
          cout<< "\n\nName:" <<name;
          cout<< "\nRegd no.:" <<regd;
          cout<<"\nBranch:"<<branch;
          cout<<"\nSem:"<<sem;
}

int main()
{
          student s;
          s.input( );
          s.display( );
}
```

OUTPUT:

```
Enter Name : Varsha
Enter Regd no.: 123
Enter Branch: CS
Enter Sem: 2

Name: Varsha
Regd no. : 123
Brach: CS
Sem: 2
```

MAKING AN OUTSIDE FUNCTION INLINE

- One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.
- We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of the function definition.

Example:

```
class  item
{
......
......
public:
void getdata(int a, float b);
};
inline void item :: getdata(int a, float b)
{
number=a;
cost=b;
}
```

NESTING OF MEMBER FUNCTION

- A member function of a class can be called only by an object of that class using a dot operator.
- However, there is an exception to this.
- A member function can be called by using its name inside another member function of the same class.

Example:

```
#include <iostream.h>
class set
{
int m, n;
public:
void input(void);
void display(void);
void largest(void);
};
int set :: largest(void)
{
if(m >= n)
return(m);
else
return(n);
}
void set :: input(void)
{
cout << "Input value of m and n"<<"\n";
cin >> m>>n;
}
void set :: display(void)
{
cout << "largest value=" << largest() <<"\n";
}

int main()
{
set A;
A.input();
A.display();

return 0;
}
```

OUTPUT:

```
Input value of m and n
25 18
Largest value=25
```

PRIVATE MEMBER FUNCTION

- Although it is normal practice to place all the data items in a private section and all the function in public, some situations may require certain function to be hidden from the outside calls.
- Tasks such a deleting an account in a customer file, or providing increment to an employee are event of serious

consequences and therefore the function handling such task should have restricted access.

- We can place these function in the private section.
- A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.

```
class sample
{
int m;
void read(void);
public:
void update(void);
void write(void);
};
```

ARRAYS WITHIN A CLASS

- Arrays can be declared as the members of a class.
- The arrays can be declared as private, public or protected members of the class.

Example:

```
#include<iostream>

const int size=5;

class student
{
int roll_no;
int marks[size];
public:
void getdata ();
void tot_marks ();
} ;

void student :: getdata ()
{
cout<<"\nEnter roll no: ";
Cin>>roll_no;
for(int i=0; i<size; i++)
{
cout<<"Enter marks in subject"<<(i+1)<<": ";
cin>>marks[i] ;
}
void student :: tot_marks() //calculating total marks
{
int total=0;
for(int i=0; i<size; i++)
total+ = marks[i];
cout<<"\n\nTotal marks "<<total;
}

void main()
student stu;
stu.getdata() ;
stu.tot_marks() ;
getch();
}
```

OUTPUT:

```
Enter roll no: 101
Enter marks in subject 1 : 67
Enter marks in subject 2 : 54
Enter marks in subject 3 : 68
Enter marks in subject 4 : 72
Enter marks in subject 5 : 82
Total  marks = 343
```

STATIC DATA MEMBER

- It is a variable which is declared with the static keyword, it is also known as class member, thus only single copy of the variable creates for all objects.
- Any changes in the static data member through one member function will reflect in all other object's member functions.

Declaration

```
static data_type member_name;
```

Defining the static data member

It should be defined outside of the class following this syntax:

```
data_type class_name :: member_name =value;
```

- If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members)

```
#include <iostream>

class Demo
{
          private:
                    static int X;

          public:
                    static void fun()
                    {
                              cout <<"Value of X: " << X << endl;
                    }
};
int Demo :: X =10;      //defining
int main()
{
          Demo X;
          X.fun();
          return 0;
}
```

```
Output:
Value of X : 10
```

Accessing static data member without static member function

- A static data member can also be accessed through the class name without using the static member , here we need an Scope Resolution Operator (SRO) **::** to access the static data member without static member function.

Syntax:

```
class_name :: static_data_member;
```

Example:

```
#include <iostream.h>
class Demo
{
          public:
                    static int ABC;
};
int Demo :: ABC =10;          //defining

int main()
{
          cout<<"\nValue of ABC: "<<Demo::ABC;
          return 0;
}
```

```
Output:
Value of ABC : 10
```

STATIC MEMBER FUNCTION IN C++

- A static member function is a special member function, which is used to access only static data members, any other normal data member cannot be accessed through static member function. Just like static data member, static member function is also a class function; it is not associated with any class object.
- We can access a static member function with class name, by using following **syntax:**

```
class_name :: function_name(parameter);
```

Example:

```
#include <iostream>
class Demo
{
          private:
                    //static data members
                    static int X;
                    static int Y;
          public:
                    //static member function
          static void  Print( )
          {
                    cout <<"Value of X: " << X << endl;
                    cout <<"Value of Y: " << Y << endl;
          }
};

//static data members initializations
int Demo :: X =10;
int Demo :: Y =20;


int main()
{
          Demo OB;
          //accessing class name with object name
          cout<<"Printing through object name:"<<endl;
          OB.Print();

          //accessing class name with class name
          cout<<"Printing through class name:"<<endl;
          Demo::Print();

          return 0;
}
```

Output:

```
Printing through object name:
Value of X : 10
Value of Y : 20
Printing through class name:
Value of X : 10
Value of Y : 20
```

ARRAY OF OBJECTS

- Like array of other user-defined data types, an array of type class can also be created.
- The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects.
- An array of objects is declared in the same way as an array of any built-in data type.

Syntax:

```
class_name array_name [size] ;
```

Example : A program to demonstrate the concept of array of objects

```
#include<iostream>
class books
{
char tit1e [30];
float price ;
public:
void getdata ();
void putdata ();
} ;
void books :: getdata ()
{
cout<<"Title:";
Cin>>title;
cout<<"Price:";
cin>>price;
 }
void books :: putdata ()
{
cout<<"Title:"<<title<< "\n";
cout<<"Price:"<<price<< "\n";
const int size=3 ;
}
void  main ( )
{
books book[size] ;
for(int i=0;i<size;i++)
{
cout<<"Enter details o£ book "<<(i+1)<<"\n";
book[i].getdata();
}
for(int i=0;i<size;i++)
{
cout<<"\nBook "<<(i+l)<<"\n";
book[i].putdata() ;
}
}
```

The output of the program is

```
Enter details of book 1
Title: c++
Price: 325
Enter details of book 2
Title: DBMS
Price:. 455
Enter details of book 3
Title: Java
Price: 255
```

```
Book 1
Title: c++
Price: 325
Book 2
Title: DBMS
Price: 455
Book 3
Title: Java
Price: 255
```

OBJECT AS FUNCTION ARGUMENTS

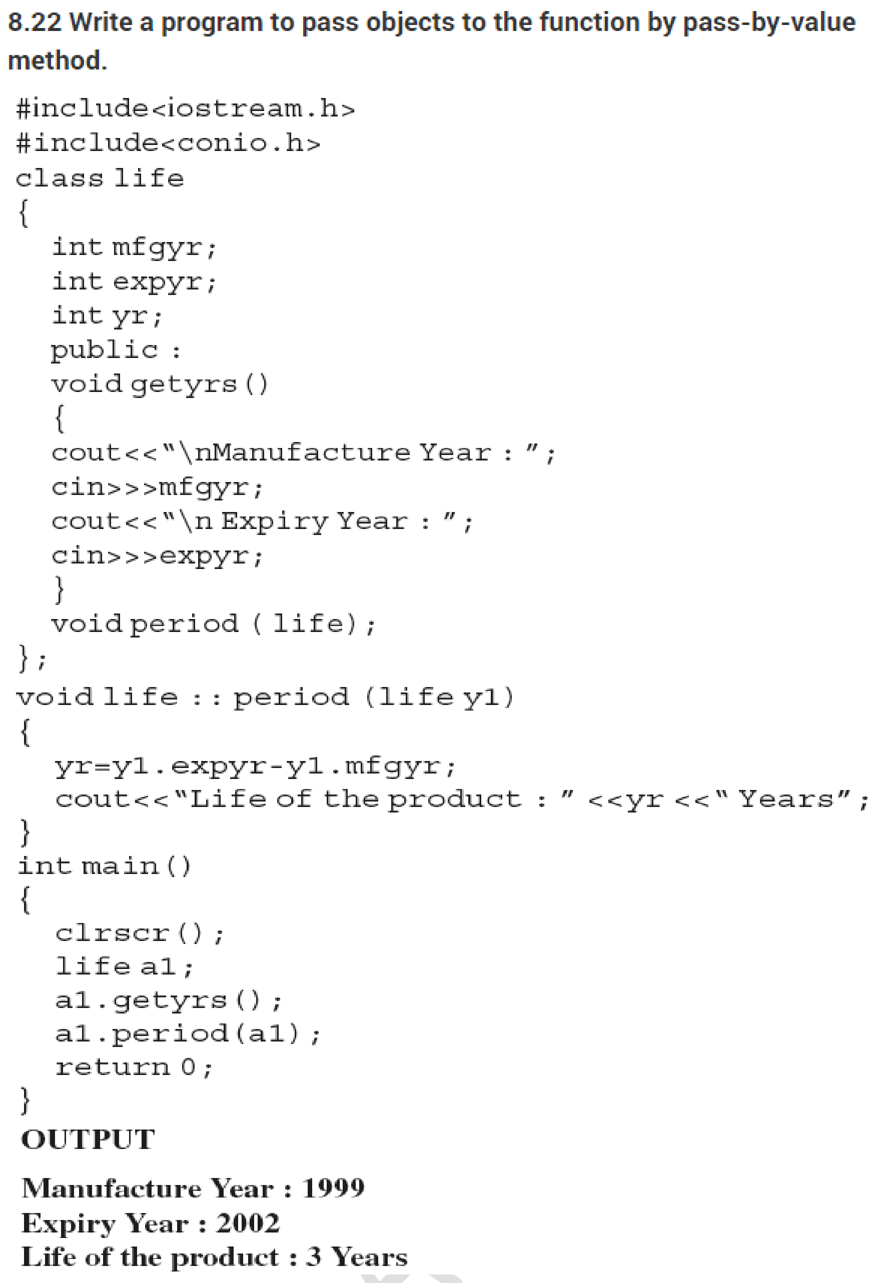
Similar to variables, object can be passed to functions. The following are the three methods to pass argument to a function:

a. Pass-by-value – A copy of object (actual object) is sent to function and assigned to the object of callee function (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object.

b. Pass-by-reference – Address of object is implicitly sent to function.

c. Pass-by-address – Address of the object is explicitly sent to function.

In pass-by-reference and pass-by-address methods, an address of actual object is passed to the function. The formal argument is reference/pointer to the actual object. Hence, changes made in the object are reflected to actual object. These two methods are useful because an address is passed to the function and duplicating of object is prevented.

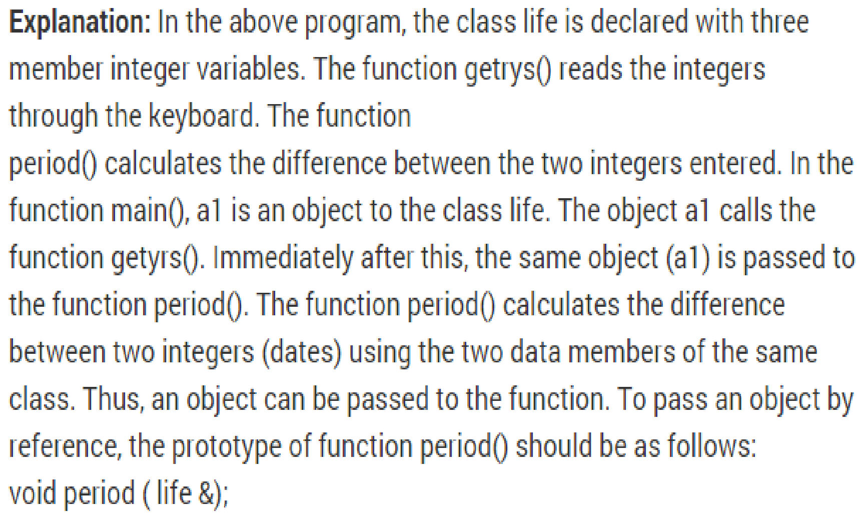


8.22 Write a program to pass objects to the function by pass-by-value method.

```
#include<iostream.h>
#include<conio.h>
class life
{
  int mfgyr;
  int expyr;
  int yr;
  public :
  void getyrs()
  {
  cout<<"\nManufacture Year : ";
  cin>>>mfgyr;
  cout<<"\n Expiry Year : ";
  cin>>>expyr;
  }
  void period ( life);
};
void life : : period (life y1)
{
  yr=y1.expyr-y1.mfgyr;
  cout<<"Life of the product : " <<yr <<" Years";
}
int main()
{
  clrscr();
  life a1;
  a1.getyrs();
  a1.period(a1);
  return 0;
}
```

OUTPUT

```
Manufacture Year : 1999
Expiry Year : 2002
Life of the product : 3 Years
```



Explanation: In the above program, the class life is declared with three member integer variables. The function getrys() reads the integers through the keyboard. The function period() calculates the difference between the two integers entered. In the function main(), a1 is an object to the class life. The object a1 calls the function getyrs(). Immediately after this, the same object (a1) is passed to the function period(). The function period() calculates the difference between two integers (dates) using the two data members of the same class. Thus, an object can be passed to the function. To pass an object by reference, the prototype of function period() should be as follows:

void period (life &);

C++ FRIEND FUNCTION

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
  // syntax of friend function.
   friend  data_type  function_name(argument/s);
 };
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Example of C++ friend function used to print the length of a box.

```
#include <iostream>
class Box
{
   private:
      int length;
   public:
      Box( ): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output:

```
Length of box : 10
```

Let's see a simple example when the function is friendly to two classes.

```
#include <iostream>

class B;         // forward declarartion.

class A
{
   int x;
   public:
   void setdata(int i)
   {
      x=i;
   }
   friend void min(A,B);        // friend function.
};

class B
{
   int y;
   public:
   void setdata(int i)
   {
      y=i;
   }
   friend void min(A,B);                  // friend function
};

void min(A a,B b)
{
   if(a.x<=b.y)
   std::cout << a.x << std::endl;
   else
   std::cout << b.y << std::endl;
}
```

```
  int main()
{
  A a;
  B b;
  a.setdata(10);
  b.setdata(20);
  min(a,b);
   return 0;
 }
```

Output:

```
10
```

POINTERS TO MEMBERS:

- It is possible to take the address of a member of a class and assign it to a pointer.
- The address of a member can be obtain by applying the operator & to a "fully qualified" class member name.
- A class member pointer can be declared using the operator **::*** with the class name.
- For example, given the class

```
class A
{
private:
int m;
public:
void show();
};
```

We can define a pointer to the member m as follows:

```
            int A::* ip= &A :: m;
```

The ip pointer created thus acts like class member in that it must be invoked with a class object.
The phrase A::* means "pointer to member of a class".
The phrase &A::m means the "address of the m member of A class".

Remember, the following statement is not valid:

```
        int *ip = &m;
```

CONST MEMBER FUNCTION:

If a member function does not alter any data in the class, then we may declare it as a const member function as follows:

```
void mul(int ,int) const;
double get_balance() const;
```

The qualifier const is appended to the function prototype. The compiler will generate an error message if such functions try to alter the data values.

CONSTRUCTORS AND DESTRUCTORS

Constructor:

- ➢ Constructor is the special type of member function in C++ classes, which are automatically invoked when an object is being created.
- ➢ It is special because its name is same as the class name.

Constructor is used for:

- **To initialize data member of class:** In the constructor member function (which will be declared by the programmer) we can initialize the default vales to the data members and they can be used further for processing.
- **To allocate memory for data member:** Constructor can also be used to declare run time memory (dynamic memory for the data members).

There are following properties of constructor:

- Constructor has the same name as the class name. It is case sensitive.
- Constructor does not have return type.
- We can overload constructor, it means we can create more than one constructor of class.
- We can use default argument in constructor.
- It must be public type.

Why constructor created?

It cannot be initialize the class member.
E.g:

```
class m
  {
     int a=10;    // illegal
  };
```

Example:

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

class sample
{          // class declaration
     int m;
  public:
     sample( );          //default constructor
        {
          m=10;
        }
     void display ( )
        {
          cout << " The value of m is " << m;
        }
 };
void main(  )
{
   sample ob;
   ob.display( );
}
```

Output:

```
Maths :  0
Science : 0
```

Types of Constructor:

- ✓ Default Constructor
- ✓ Parameterize Constructor
- ✓ Copy Constructor

Default Constructor:

- ➢ Default constructor is also known as zero argument constructors.
- ➢ Default constructor does not have any parameters and is used to set (initialize) class data members.
- ➢ Since, there is no argument used in it, it is called **"Zero Argument Constructor"**.
- ➢ In a class, if there is no default constructors defined, then the compiler inserts a default constructor with an empty body in the class in compiled code.
- ➢ Constructor can also be defined outside of the class; it does not have any return type.

Syntax

```
class_name() {
-----
-----
}
```

Example of Default Constructor

Let us take the example of class `Marks` which contains the marks of two subjects Maths and Science.

```
#include<iostream.h>

class Marks
{
public:
  int maths;
  int science;

  //Default Constructor
  Marks( ) {
    maths=60;
    science=80;
  }

  display() {
     cout << "Maths :  " << maths <<endl;
     cout << "Science :" << science << endl;
   }
};

int main() {
  //invoke Default Constructor
  Marks m;
  m.display();
  return 0;
}
```

Parameterized Constructors:

- ➢ It may be necessary to initialize the various data elements of different objects with different values when they are created.
- ➢ This is achieved by passing arguments to the constructor function when the objects are created.
- ➢ The constructors that can take arguments are called parameterized constructors.

Syntax

```
class_name( Argument_List)
{
-----
-----
}
```

Example of Parametrized Constructor

Let us take the example of class ‘Marks’ which contains the marks of two subjects Maths and Science.

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

class Marks
{
public:
  int maths;
  int science;

  Marks(int mark1,int mark2)
//Parametrized Constructor
{
     maths = mark1;
     science = mark2;
  }

  void display() {
     cout << "Maths :  " << maths <<endl;
     cout << "Science :" << science << endl;
  }
};

int main( ) {
   Marks m(90,85);  //invoke Parametrized Constructor
   m.display();
  return 0;
}
```

Output:

```
Maths :  90
Science  :  85
```

Copy Constructor:

- ➢ The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

The copy constructor is used to −

1. Initialize one object from another of the same type.
2. Copy an object to pass it as an argument to a function.
3. Copy an object to return it from a function.

- ➢ If a copy constructor is not defined in a class, the compiler itself defines one.

- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

Syntax

```
class_name(class_name  &object )
{
-----
-----
}
```

Example:

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

class copy
{                        // class declaration
       int m;
   public:
       copy( );           //default constructor
         {

         }
       copy(  int x);          //parameterized constructor
         {
              m= x;
          }
       copy(  copy  &c);          //copy constructor
         {
              m= c.m;
          }

  };

void main(  )
{
   copy c1(25);
   copy  c2(c1);
   copy  c3;
   c3 = c1;

    cout<<c1.a<<endl;
    cout<<c2.a<<endl;
    cout<<c3.a<<endl;
}
```

Output:

```
23
23
23
```

MULTIPLE CONSTRUCTOR IN A CLASS (CONSTRUCTOR OVERLOADING):

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Example:

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

class construct
{
public:
   float area;

   construct( )          // Constructor with no parameters
   {
      area = 0;
```

Output:

```
0
200
```

```
    }
    construct(int a, int b)       // Constructor with two parameters
    {
        area = a * b;
    }
    void disp()
    {
        cout<< area<< endl;
    }
};

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

DYNAMIC CONSTRUCTOR

- Dynamic constructor is used to allocate the memory to the objects at the run time.
- Memory is allocated at run time with the help of 'new' operator.
- By using this constructor, we can dynamically initialize the objects.

Example:

```
#include <iostream.h>
#include <conio.h>
class dyncons
{
 int * p;
public:
dyncons()
{
 p=new int;
 *p=10;
}
dyncons(int v)
{
 p=new int;
 *p=v;
}
int dis()
{ return(*p);
}
};

void main()
{
clrscr();
dyncons o, o1(9);
cout<<"The value of object o's p is:";
cout<<o.dis();
cout<<"\nThe value of object 01's p is:"<<o1.dis();
getch( );
}
```

Output:

```
The value of object o’s p is :10
The value of object o1’s p is : 9
```

DESTRUCTOR

- Constructor allocates the memory for an object.
- Destructor deallocate the memory occupied by an object.
- Like constructor, destructor name and class name must be same, preceded by a tilde(~) sign.
- Destructor take no argument and have no return value.
- Constructor is invoked automatically when the

object created.

- ➢ Destructor is invoked when the object goes out of scope.
- ➢ In other words, Destructor is invoked, when compiler comes out form the function where an object is created.

Syntax:

```
class   A
{
   public:
         ~A( )      // defining destructor  for class
         {
                  // statement
         }
};
```

C++ Destructor Program #1 : Simple Example

```
#include<iostream.h>
class Marks
{
public:
  int maths;
  int science;
  Marks( )              //constructor
{
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }
  ~Marks(  )      //Destructor
 {
    cout << "Inside Destructor"<<endl;
    cout << "C++ Object destructed"<<endl;
  }
};
void main( )
{
  Marks m1;
  Marks m2;
}
```

Output

```
Inside Constructor
C++ Object created
Inside Constructor
C++ Object created

Inside Destructor
C++ Object destructed
Inside Destructor
C++ Object destructed
```

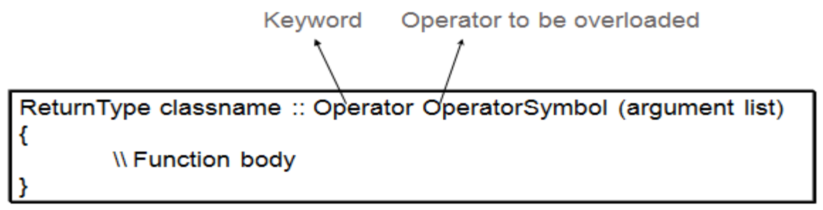
OPERATOR OVERLOADING IN C++

- ➢ Operator overloading is an important concept in C++.
- ➢ It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.
- ➢ Overloaded operator is used to perform operation on user-defined data type.
- ➢ For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- ✓ scope operator - ::
- ✓ sizeof
- ✓ member selector - .
- ✓ member pointer selector - *
- ✓ ternary operator - ?:

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.
An operator can be overloaded by defining a function to it. The function for operator is declared by using the **operator** keyword followed by the operator.

There are two types of operator overloading in C++

- **Binary Operator Overloading**
- **Unary Operator Overloading**

<u>Overloading Unary Operator</u>

- ➢ Unary operator is an operator that takes single operand(variable).
- ➢ Both increment(++) and decrement(--) operators are unary operators.

Example of Unary Operator Overloading

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

        class Rectangle
        {
            int L,B;
         public:
                Rectangle()        //Default Constructor
                {
                L = 0;
                B = 0;
                }
        void operator++( )  // Unary operator overloading func.
        {
                B+=2;
        }
        void Display( )
        {
                cout<<"\n\tLength : "<<L;
                cout<<"\n\tBreadth : "<<B;                    }

    };
        void main()
        {
                Rectangle R;        //Creating Object


        cout<<"\n\tLength Breadth before increment";
        R.Display();
        R++;
        cout<<"\n\n\tLength Breadth after increment";
        R.Display();
        }
```

Output :

```
        Length Breadth after increment
        L : 0
        B : 0

        Length Breadth after increment
        L : 2
        B : 2
```

<u>Overloading Binary Operator</u>

- ➢ Binary operator is an operator that takes two operand(variable).

- ➢ Binary operator overloading is similar to unary operator overloading except that a binary operator overloading requires an additional parameter.

Binary Operators

- Arithmetic operators (+, -, *, /, %)
- Arithmetic assignment operators (+=, -=, *=, /=, %=)
- Relational operators (>, <, >=, <=, !=, ==)

Example of Binary Operator Overloading

```
        #include<iostream.h>
        #include<conio.h>
        class Rectangle
        {
                int  L, B;
           public:
                Rectangle()          //Default Constructor
                {
                        L = 0;
                        B = 0;
                }
                Rectangle(int x,int y)    //Parameterize Constructor
                {
                        L = x;
                        B = y;
                }
     //Binary operator overloading func.
        Rectangle operator+(Rectangle Rec)                    {
                        Rectangle R;

                                R.L = L + Rec.L;
                                R.B = B + Rec.B;
                        return R;
                }
                void Display()
                {
                        cout<<"\n\tLength : "<<L;
                        cout<<"\n\tBreadth : "<<B;
                }
        };

        void main()
        {
          Rectangle R1(2,5),R2(3,4),R3;      //Creating Objects
          cout<<"\n\tRectangle 1 : ";
          R1.Display();

          cout<<"\n\n\tRectangle 2 : ";
          R2.Display();

          R3 = R1 + R2;                  //  Statement 1

          cout<<"\n\n\tRectangle 3 : ";
          R3.Display();
        }
```

Output :

```
 Rectangle 1 :
 L : 2
 B : 5

 Rectangle 2 :
 L : 3
 B : 4

 Rectangle 3 :
 L : 5
 B : 9
```

In statement 1, Left object R1 will invoke operator+() function and right object R2 is passing as argument.

Another way of calling binary operator overloading function is to call like a normal member function as follows,

```
R3 = R1.operator+ ( R2 );
```

OVERLOAD BINARY OPERATOR USING FRIEND FUNCTION

- If you define operator function as a **friend function** then it will accept **two arguments**.
- Because friend functions is not a member function so it is not invoked using object of the class.
- Thus we need to pass two objects as an argument explicitly.
- Consider following example to overload binary operator + using friend function.

```
#include <iostream.h>
class demo
{
int x,y;
public:
demo()
{
x=0;
y=0;
}
demo(int a, int b)
{
x=a;
y=b;
}
friend demo operator + (demo &d1, demo &d2)
{
demo d3;
d3.x = d1.x + d2.x;
d3.y = d1.y + d2.y;
return d3;
}
void display()
{
cout<<"X="<<x<<endl;
cout<<"Y="<<y<<endl;
}
};
int main()
{
demo d1(2,3);
demo d2(4,5);
demo d3;
d3 = operator + (d1,d2);

cout<<"Object C1\n";
d1.display();
cout<<"Object C2\n";
d2.display();
cout<<"Object C3\n";
d3.display();
return 0;
}
```

Output:

```
Object C1
X=2
Y=3
Object C2
X=4
Y=5
Object C3
X=6
Y=8
```

STRING MANIPULATION USING OPERATOR OVERLOADING

- C++ allows us the facility of manipulate strings using the concept of operator overloading. **For example** we can overload **+** operator to **concate** two strings. We can overload **==** operator to **compare** two strings.
- Consider Following Example in which we overload + operator to concate two strings.

```
#include<iostream.h>
#include<string.h>
class string
```

```
{
char *name;
int length;
public:
string()
{
length=0;
name = new char[length+1];
}
string(char *n)
{
length=strlen(n);
name= new char [length+1];
strcpy(name,n);
}
void display()
{
cout<<"String:"<<name;
}
string operator+(string s)
{
string temp;
strcpy(temp.name,name);
strcat(temp.name,s.name);
return temp;
}
};
int main()
{
string s1("Hello");
string s2("Welcome");
string s3;
s1.display();
s2.display();
s3=s1+s2;
s3.display();
return 0;
}
```

RULES OF OPERATOR OVERLOADING IN C++

Every programmer knows the concept of operation overloading in C++. Although it looks simple to redefine the operators in operator overloading, there are certain restrictions and limitation in overloading the operators. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be overloaded.

2. The overloaded operator must have at least one operand that is of user defined type.

3. We cannot change the basic meaning of an operator. That is to say, We cannot redefine the plus(+) operator to subtract one value from the other.

4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.

5. There are some operators that cannot be overloaded like size of operator(sizeof), membership operator(.), pointer to member operator(.*), scope resolution operator(::), conditional operators(?:) etc

6. We cannot use “friend” functions to overload certain operators.However, member function can be used to overload them. Friend Functions can not be used with assignment operator(=), function call operator(()), subscripting operator([]), class member access operator(->) etc.

7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevent class).

8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

10. Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

TYPE CONVERSION

WHAT IS TYPE CONVERSIONS?

- ✓ Converting data from one type into another type is called type conversion.
- ✓ In other words converting an expression of a given type into another is called type casting (by programmer).

Eg.

```
int m;
float =3.14;
m=x;        // conversion of float type data into integer type
```

Types Of Type Conversion:

Four type of situation might arise in the data conversion.

1. Conversion from basic type to basic type
2. Conversion from basic type to class type
3. Conversion from class type to basic type
4. Conversion from class type to class type

1. **Conversion from basic type to basic type**

There are two ways of achieving the type conversion namely:

- Automatic Conversion otherwise called as **Implicit Conversion**
- Type casting otherwise called as **Explicit Conversion** (by programmer).

Implicit conversion:

- ✓ This is not done by any conversions or operators.
- ✓ In other words the value gets automatically converted to the specific type to which it is assigned.

Example:

```
#include <iostream.h>
void main()
{
float x=60.68;
int y;
y=x;   // implicit type conversion of float to integer
}
```

Explicit Conversion:

- ✓ **Type casting otherwise called as Explicit Conversion**
- ✓ Explicit conversion can be done using type cast operator.

Syntax:

```
datatype (expression);
```

Here in the above datatype is the type which the programmer wants the expression to gets changed as.

In C++ the type casting can be done in either of the two ways mentioned below namely:

- C-style casting : (type) expression
- C++-style casting : type (expression)

Example:

```
#include <iostream.h>
void main()
{
int a;
float b,c;
cout << "Enter the value of a:";
cin >> a;
cout << "Enter the value of b:";
cin >> b;
c = float(a)+b;
cout << "The value of c is:" << c;
}
```

2. Basic type to class type:

Convertion of basic type to class type can be done using constructor.
Eg.

```
#include <iostream.h>
class hours
{
            int hrs;
public:
 hours (int t)
{
   hrs = t/60;
 }
void show( )
{
   cout<<hrs<<":hours"<<endl;
}
};

void main()
{
       hours t1 = 85;   // integer 85 is converted object of class
       hours
       t1.show( );    // output    1 : hours
}
```

3. Class type to basic type:

✓ It can be done by using: overloaded casting operator
✓ It is also known as conversion function.

Syntax:

```
operator float( )
{
        return(basic_type data);
}
```

The casting operator function should satisfy following conditions:

- It must be a class member
- It must not specify return type
- It must not have any argument.

Example:

```
class hours {
int hrs;
public:
hours ( int t)
{
hrs = t/60;
}
operator float( )   // type conversion from class to basic
{
   return float (hrs)/2;        // type conversion int to float (basic to basic)
}
void show ( )
{
          cout<<hrs<<":hours"<<endl;
}
};
void main ( )
{
float  f;
hours t1 =85;    // integer 85 is converted into objects of class hours
t1.show( );                  // output:  1: hours
f=t1;                     // type conversion from class objects " t1" to float " f"
cout<<"time:"<< f <<endl;    // output:   time:  0.5
}
```

4. Class Type To Class Type:

Obx=oby;

Here class y is source class and class X is destination class.
There are two ways to convert class type to class type conversion.

1. Using casting operator function in destination class. (or)

2. Using constructor conversion in source class.

In this type of conversion both the type that is source type and the destination type are of class type. Means the source type is of class type and the destination type is also of the class type. In other words, one class data type is converted into another class type.
For example we have two classes one for "*computer"* and another for "*mobile"*. Suppose if we wish to assign *"price"* of *computer* to *mobile* then it can be achieved by the statement below which is the example of the conversion from one class to another class type.

```
mob = comp ;
// where mob and comp are the objects of mobile and computer classes respectively.
```

- Here the assignment will be done by converting *"comp"* object which is of class type into the *"mob"* which is another class data type.

Example:
1.Class type using casting operator function in source class

```
// source class
class minutes
{
        int m;
public:
        minutes ( int ms)
{
m=ms;
}
operator hours( )
{
hours h1;
h1.h = m/60;
return(h1);
}

void show ( )
{
cout<<"minutes = " << m << endl;
}
int getdata( )
{
return m;
}
};
```

```
//destination class
 class hours
{
int h;
public:
hours ( )
{
h = 0;
}
void show ( )
{
cout<<"hours ="<<h <<endl;
}
};
```

use of class type to class type conversion in main( )

```
int main( )
{
minutes min(60);
hours hr;               // output :   minutes  = 60
hr=min;                 //class minutes to class hours
min.show( );                    // output :   minutes  = 60
hr.show ( );            // output:  hours =1
getch( );
}
```

2. Class type to class type using constructor conversion in destination class

class minutes	class hours
{	{
int m;	int h;
public:	public:
minutes (int ms)	hours ( )
{	{
m=ms;	h = 0;
}	}
void show ( )	void show ( )
{	{
cout<<”minutes = “ << m	cout<<”hours =”<<h <<endl;
<< “\n”;	}
}	**hours h1;**
int getdata( )	**{**
{	**h1.h = m/60;**
return m;	**return(h1);**
}	**}**
};	};

UNIT - III
INHERITANCE

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.
  Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
  Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.
- The main advantages of inheritance are **code reusability** and **readability**. When child class inherits the properties and functionality of parent class, we need not to write the same code again in child class. This makes it easier to reuse the code, makes us write the less code and the code becomes much more readable.

The general form of defining a derived class is:

```
class derived-class_name : visibility-mode base-class_name
{
 . . . .  // members of the derived class
 . . . .
};
```

Modes of Inheritance

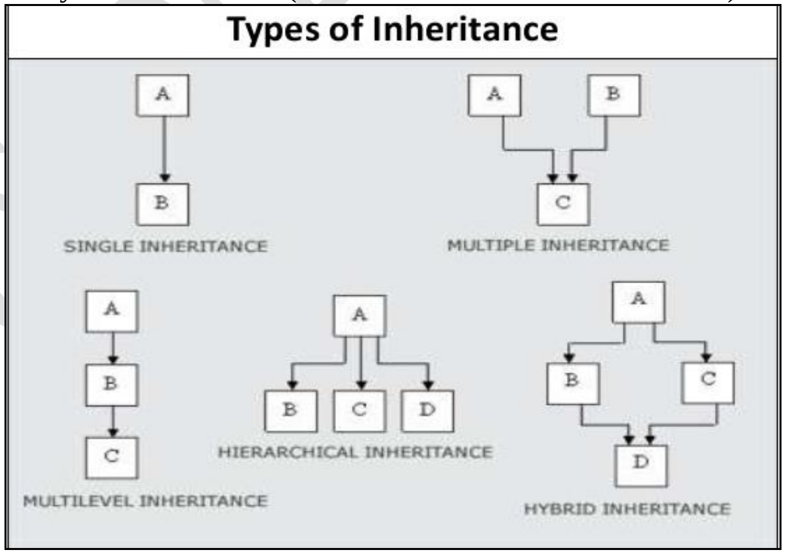
1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

Types of Inheritance:

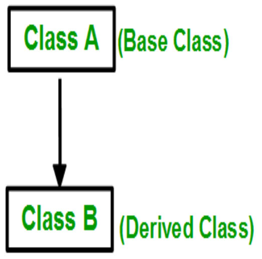
C++ offers five types of Inheritance. They are:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)



Single Inheritance:

- ✓ In single inheritance, there is only one base class and one derived class.
- ✓ The Derived class gets inherited from its base class.
- ✓ This is the simplest form of inheritance.



Syntax:

```
class base_class
{
};
class derived_ class : visibility-mode base_ class
{
};
```

```
#include <iostream>
 class Animal
{
  public:
 void eat()
   {
      cout<<"Eating..."<<endl;
    }
  };
  class Dog: public Animal
  {
      public:
    void bark(){
   cout<<"Barking...";
    }
   };
int main(void) {
   Dog d1;
   d1.eat();
   d1.bark();
   return 0;
}
```

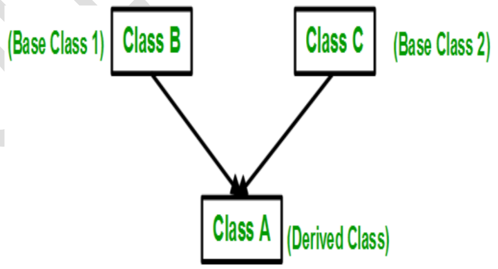
```
Output:
Eating...
Barking...
```

Ambiguity in single Inheritance

- ✓ Whenever a data member and member functions are defined with the same name in both the base and derived class, ambiguity occurs.
- ✓ The scope resolution operator must be used to refer to particular class as: object_name.class_name :: class member.

Multiple Inheritance:

- ➢ Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.
- ➢ The process in which a derived class inherits its from several base classes, is called multiple inheritance.
- ➢ In Multiple inheritance, there is only one derived class and several base classes.

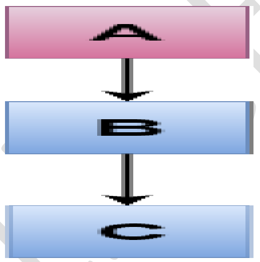


Syntax:

```
class base_class1{
};
class base_class2{
};
class derived_ class : visibility-mode base_ class1 , visibility-mode
base_ class2 {
};
```

```
#include <iostream>  class A
        int a;
     public:
     void get_a(int n)
     {
        a = n;
     }
class B
 {
     int b;
     public:
     void get_b(int n)
      {
         b = n;
      }
  };
  class C : public A,public B
  {
     public:
      void display()
     {
        cout << "The value of a is : " <<a<< endl;
        cout << "The value of b is : " <<b<<  endl;
        cout<<"Addition of a and b is : "<<a+b;
     }
 };
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

     return 0;
}
```

Multilevel Inheritance:



- The process in which a derived class inherits traits from another derived class, is called Multilevel Inheritance.
- A derived class with multilevel inheritance is declared as :

Syntax:

```
class base_class {
};
class derived_ class1 : visibility-mode base_ class {
};
class derived_ class 2: visibility-mode derived_ class1 {
};
```

Here, derived_ class 2 inherits traits from derived_ class 1 which itself inherits from base_class.

```
#include <iostream>
 class Animal {
  public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
  };
  class Dog: public Animal
  {
     public:
    void bark(){
   cout<<"Barking..."<<endl;
    }
  };
  class BabyDog: public Dog
  {
     public:
```

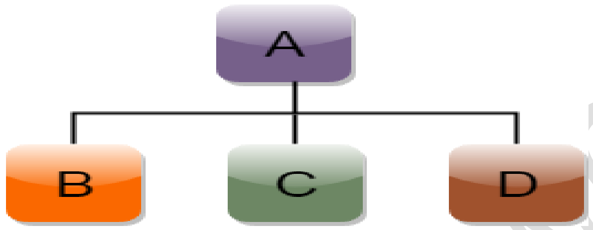
Output:

```
Eating...
Barking...
Weeping...
```

```
    void weep() {
   cout<<"Weeping...";
    }
  };
void main(void) {
   BabyDog d1;
   d1.eat();
   d1.bark();
    d1.weep();
}
```

Hierarchical Inheritance:

- ➢ The process in which traits of one class can be inherited by more than one class is known as Hierarchical inheritance.
- ➢ The base class will include all the features that are common to the derived classes.
- ➢ A derived class can serve as a base class for lower level classes and so on.



Syntax of Hierarchical inheritance:

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Example:

```
#include <iostream>
class Shape                  // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;
   }
};
```

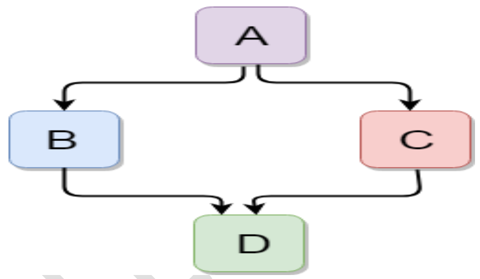
```
void  main()
{
   Rectangle r;
   Triangle t;
   int length,breadth,base,height;
   cout << "Enter the length and breadth of a rectangle: " <<  endl;
   cin>>length>>breadth;
   r.get_data(length,breadth);
   int m = r.rect_area();
   cout << "Area of the rectangle is : " <<m<<  endl;
   cout << "Enter the base and height of the triangle: " <<  endl;
   cin>>base>>height;
   t.get_data(base,height);
   float n = t.triangle_area();
   cout <<"Area of the triangle is : "  << n<< endl;
   }
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

Hybrid Inheritance

- The inheritance hierarchy that reflects any legal combination of other types of inheritance is known as hybrid Inheritance.
- Hybrid inheritance is a combination of more than one type of inheritance.
- For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.



Example:

```
#include <iostream>
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
    cout << "Enter the value of 'a' : " <<  endl;
     cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      cout << "Enter the value of 'b' : " <<  endl;
     cin>>b;
   }
};
class C
{
   protected:
   int c;
```

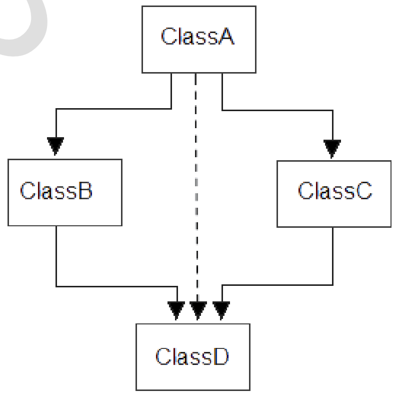
```
  public:
  void get_c()
  {
   cout << "Enter the value of c is : " <<  endl;
     cin>>c;
  }
};

class D : public B, public C
{
   protected:
   int d;
   public:
   void mul()
   {
        get_a();
        get_b();
        get_c();
        cout << "Multiplication of a,b,c is : " <<a*b*c<<  endl;
   }
};
int main()
{
   D d;
   d.mul();
   return 0;
}
```

Output:

```
Enter the value of 'a' :  10
Enter the value of 'b' :    20
Enter the value of c is :   30
Multiplication of a,b,c is : 6000
```

VIRTUAL BASE CLASSES

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class.
- This can be achieved by preceding the base class' name with the word virtual.
- Virtual base class is used in situation where a derived have multiple copies of base class.
- Consider the following figure:



Syntax:

```
class A
{
………..
………..

 };
class B : public virtual A
{
………..
………..

};
class C : public virtual A
{
```

```
……...
……...
};
class D : public B, public C
{
……...
……...
};
```

Example using virtual base class

```
#include<iostream.h>
#include<conio.h>
class ClassA
{
      public:
      int a;
};
class ClassB : virtual public ClassA
{
      public:
      int b;
};
class ClassC : virtual public ClassA
{
      public:
      int c;
};
class ClassD : public ClassB, public ClassC
{
      public:
      int d;
};

void main()
{
            ClassD obj;

            obj.a = 10;        //Statement 1
            obj.a = 100;       //Statement 2

            obj.b = 20;
            obj.c = 30;
            obj.d = 40;

            cout<< "\n A : "<< obj.a;
            cout<< "\n B : "<< obj.b;
            cout<< "\n C : "<< obj.c;
            cout<< "\n D : "<< obj.d;
}
```

Output :

```
A : 100
B : 20
C : 30
D : 40
```

According to the above example, **ClassD** have only one copy of **ClassA** and statement 4 will overwrite the value of **a**, given in statement 3.

C++ Abstract Class

- Abstract class is used in situation, when we have partial set of implementation of methods in a class.
- ***For example***, consider a class have four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use **abstract class.**
- A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

- A class with at least one **pure virtual function** or **abstract function** is called abstract class.
- Pure virtual function is also known as abstract function.

  - We can't create an object of abstract class b'coz it has partial implementation of methods.
  - Abstract function doesn't have body
  - We must implement all abstract functions in derived class.

C++ abstract class : syntax and structure

```
//declaring abstract base class
class base_class
 {
  virtual return_type func_name() = 0; //pure virtual function
 }
```

Example of C++ Abstract class

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

class BaseClass        //Abstract class
{
       public:
        virtual void Display1( )=0;     //Pure virtual func or abstract func
        virtual void Display2( )=0;     //Pure virtual func or abstract func
        void Display3( )
        {
              cout<<"\n\tThis is Display3( ) method of Base Class";
        }
};

class DerivedClass : public BaseClass
{
       public:
        void Display1( )
        {
              cout<<"\n\tThis is Display1( ) method of Derived Class";
        }
        void Display2( )
        {
              cout<<"\n\tThis is Display2( ) method of Derived Class";
        }
};

void main()
{

       DerivedClass D;

       D.Display1( );      // This will invoke Display1() method of Derived Class
       D.Display2( );      // This will invoke Display2() method of Derived Class
       D.Display3( );      // This will invoke Display3() method of Derived Class
}
```

Output :

```
       This is Display1( ) method of Derived Class
       This is Display2( ) method of Derived Class
       This is Display3( ) method of Base Class
```

CONSTRUCTORS IN DERIVED CLASS

When a class is declared, a constructor is also declared inside the class in order to initialize data members. It is not possible to use a single constructor for more classes. Every class has its own constructor and destructor with a similar name as the class. When a class is derived from another class, it is possible to define a constructor in the derived class, and the data members of both base and derived classes can be initialized. It is not essential to declare a constructor in a base class. Thus, the constructor of the derived class works for its base class; such constructors are called constructors in the derived class or common constructors.

11.29 Write a program to initialize member variables of both base and derived class using a constructor of derived class.

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

class A
{
   protected:
   int x;
   int y;
};
class B : private A
{
   public:
   int z;
   B() {x=1,y=2,z=3;
   cout<<"x="<<x <<"y="<<y <<"z="<<z;}
};
int main()
{
   clrscr();
   B b;
   return 0;
}
```

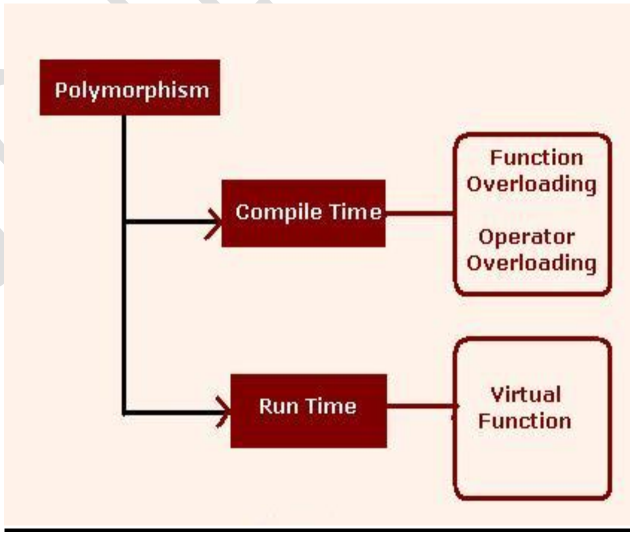
OUTPUT

```
x=1 y=2 z=3
```

Explanation: In the above program, the classes A B are declared. The class B is derived from the class A. The constructor of the class B initializes member variables of both classes. Hence, it acts as a common constructor of both base and derived classes.

POLYMORPHISM IN C++

Polymorphism is the technique of using same thing for different purpose.
There are two types of polymorphism:
(A) Compile time polymorphism
(B) Run time polymorphism



(A) Compile time polymorphism:

Compile time polymorphism is also known as static binding or early binding.
Function overloading and **Operator overloading** are the example of compile time polymorphism.
It is called compile time polymorphism because which version of function to invoke is determined by the compiler at compile time based on number and types of the argument.
Thus in compile time polymorphism which function to invoke

is determined at compile time so it is called **static or early binding**.

(B) Run time polymorphism:

Run time polymorphism is also known as **dynamic binding or late binding**.

Virtual function is the example of run time polymorphism.

While inheriting derived class from base class if both classes contain same function then we have to declare that function as a virtual in the base class.

In order to invoke function from the appropriate class you need to declare a pointer of base class and then invoke the function using that pointer. If pointer contains the address of the base class object then base class version is invoked and if pointer contains the address of derive class object then derived class version is invoked.

Thus in run time polymorphism which function to invoke is determined at runtime so it is called **dynamic binding or late binding**.

POINTERS

What are Pointers?

- A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.
- The general form of a pointer variable declaration is

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable.

The asterisk you used to declare a pointer is the same asterisk that you use for multiplication.

However, in this statement the asterisk is being used to designate a variable as a pointer.

Following are the valid pointer declaration −

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

Declaration of Pointer variable:

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip      // pointer to integer variable
float *fp;      // pointer to float variable
double *dp;     // pointer to double variable
char *cp;       // pointer to char variable
```

Initialization of Pointer variable:

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>
void main( )  {
   int a = 10;
   int *ptr;     //pointer declaration
   ptr = &a;      //pointer initialization
 }
```

Pointer variable a always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>
void main()
{
   float a;
   int *ptr;
   ptr = &a;      // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULLvalue is called a **NULL pointer**.

```
#include <stdio.h>

int main()
{
   int *ptr = NULL;
   return 0;
}
```

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

POINTERS TO OBJECTS

- A variable that holds an address value is called a pointer variable or simply pointer.
- Pointer can point to objects as well as to simple data types and arrays. sometimes we don't know, at the time that we write the program , how many objects we want to create.
- When this is the case we can use new to create objects while the program is running. new returns a pointer to an unnamed objects.

Example:

```
#include <iostream>
#include <string>

class student
{
private:
        int rollno;
        string name;
public:
        student():rollno(0),name("")
        {}
        student(int r, string n): rollno(r),name (n)
        {}
        void get()
        {
                cout<<"enter roll no";
                cin>>rollno;
                cout<<"enter  name";
                cin>>name;
        }
        void print()
        {
                cout<<"roll no is "<<rollno;
                cout<<"name is "<<name;
            }
    };
void main ()
{
        student *ps=new student;
        (*ps).get();
        (*ps).print();
        delete ps;
}
```

C++ THIS POINTER

- C++ provides a keyword 'this', which represents the current object and passed as a hidden argument to all member functions.
- The **this** pointer is a constant pointer that holds the memory address of the current object.
- The **this** pointer is not available in static member functions as static member functions can be called without any object. Static member functions can be called with class name.

Example of this pointer

```
        #include<iostream.h>
        #include<conio.h>
        #include<string.h>
        class Student
        {
          int Roll;
          char Name[25];
          float Marks;

          public:
             Student(int R,float Mks,char Nm[ ])        //Constructor 1
                 {
                 Roll = R;
                 strcpy(Name,Nm);
                 Marks = Mks;
                 }

        Student(char Name[],float Marks,int Roll)      //Constructor 2
              {
                Roll = Roll;
                strcpy(Name,Name);
                Marks = Marks;
              }

        Student(int Roll,char Name[],float Marks)      //Constructor 3
                 {
                      this->Roll = Roll;
                      strcpy(this->Name,Name);
                      this->Marks = Marks;
                 }
          void Display()
                 {
                         cout<<"\n\tRoll : "<<Roll;
                         cout<<"\n\tName : "<<Name;
                         cout<<"\n\tMarks : "<<Marks;
                 }
          };

          void main(  )
          {
                 Student S1(1,89.63,"Sumit");
                 Student S2("Kumar",78.53,2);
                 Student S3(3,"Gaurav",68.94);

                 cout<<"\n\n\tDetails of Student 1 : ";
                 S1.Display();

                 cout<<"\n\n\tDetails of Student 2 : ";
                 S2.Display();
                cout<<"\n\n\tDetails of Student 3 : ";
                S3.Display();
        }
```

```
        Output :

        Details of Student 1 :
        Roll : 1
        Name : Sumit
        Marks : 89.63

        Details of Student 2 :
        Roll : 31883
        Name : ?&;6 • #?#?6 • #N$?%_5$?
        Marks : 1.07643e+24

        Details of Student 3 :
        Roll : 3
        Name : Gaurav
        Marks : 68.94
```

In constructor 1,variables declared in argument list different from variables declared as class data members. When compiler doesn't find Roll, Name, Marks as local variable, then, it will find Roll, Name, Marks in class scope and assign values to them.

But Constructor 2 will not initialize class data members. When we pass values to constructor 2, it will initialize values to itself local variables b'coz variables declared in argument list and variable declared as data members are of same name.

In this situation, we use 'this' pointer to differentiate local variable and class data members as shown in constructor 3.

POINTER TO DERIVED CLASS OBJECT

- In C++ you can declare a pointer that contains the address of the object of type class.
- Suppose we have created a class named base as shown below:

```
class Base   {
public:
int x;
void display ()
{
cout<<"X="<<x<<endl;
}
};
```

Now you can declare a pointer that contains address of the base class object as shown below:

```
Base *ptr; // declare a pointer of base class
Base B1; // declare an object of base class
Ptr = &B1; // assign address of object to base class pointer
```

Using this pointer you can access members of the base class as shown below:

```
ptr->x = 10;
ptr->display ();
```

Now derived a new class named Derive from base class as shown below:

```
class Derive: public Base
{
public:
int y;
void display ();
{
```

cout<<"X="<<x<<endl;
cout<<"Y="<<y<<endl;
}
};

C++ allows you to assign the address of the derived class object to the base class pointer as shown below:

Derive D1; // declare an object of derived class
ptr = &D1; // assign address of derive class object to base class pointer.

Now if you try to access the member of derived class using base class pointer it will not allow you to access the member of derived class as shown below:

ptr->y = 20; // It will generate an error
ptr->display (); // It will invoke the display() of base class

Thus to overcome this problem you have to declare a pointer of derived class and then assign the address of derived class object to this pointer as shown below:

Derive *ptr1;
Derive D1;
***ptr1 = &D1;**
ptr1->y = 20;
ptr1->display (); // it will invoke the display () of derived class.

Thus in order to access member of particular class you need to create a pointer of that class and then assign the address of that class object to the pointer.

```
#include <iostream.h>
class Base    {
        public:
             int x;
             void display ()
             {
                     cout<<"X="<<x<<endl;
             }
};
class Derive: public Base
{
             public:
             int y;
             void display ();
             {
                     cout<<"X="<<x<<endl;
                     cout<<"Y="<<y<<endl;
             }
};
int main ()
{
             Base B1;
             Base *ptr;
             ptr = &B1;
             ptr->x = 10;
             ptr->display();
             Derive D1;
             Derive *ptr1;
             ptr1 = &D1;
             ptr1->x = 10;
             ptr1->y = 20;
             ptr1->display ();
}
```

```
Output:
X= 10
X = 10
Y = 20
```

C++ VIRTUAL FUNCTION

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage:

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

Example of virtual function

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

class BaseClass
{
    public:
     virtual void Display()
     {
          cout<<"\n\tThis is Display() method of Base Class";
     }

     void Show()
     {
          cout<<"\n\tThis is Show() method of Base Class";
     }

};

class DerivedClass : public BaseClass
{
    public:
     void Display()
     {
          cout<<"\n\tThis is Display() method of Derived Class";
     }
```

```
        void Show()
        {
              cout<<"\n\tThis is Show() method of Derived Class";
        }
};
void main()
{
      DerivedClass D;
      BaseClass *B;          //Creating Base Class Pointer
      B = new BaseClass;

      B->Display();      //This will invoke Display() method of Base Class
      B->Show();           //This will invoke Show() method of Base Class

      B=&D;

      B->Display();        //This will invoke Display() method of Derived Class
                             //bcoz Display() method is virtual in Base Class
      B->Show();       //This will invoke Show() method of Base Class
                      //bcoz Show() method is not virtual in Base Class
}
```

Output :

```
      This is Display() method of Base Class
      This is Show() method of Base Class
      This is Display() method of Derived Class
      This is Show() method of Base Class
```

PURE VIRTUAL FUNCTION IN C++

- A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.
- Pure virtual function doesn't have body or implementation.
- We must implement all pure virtual functions in derived class.
- Pure virtual function is also known as abstract function.
- A class with at least one pure virtual function or abstract function is called abstract class.
- We can't create an object of abstract class.
- Member functions of abstract class will be invoked by derived class object.

Example of pure virtual function

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

   class BaseClass       //Abstract class
   {
         public:
          virtual void Display1( )=0;     //Pure virtual func or abstract func
          virtual void Display2( )=0;     //Pure virtual func or abstract func
          void Display3( )
          {
                cout<<"\n\tThis is Display3( ) method of Base Class";
          }
   };

   class DerivedClass : public BaseClass
   {
         public:
          void Display1( )
          {
                cout<<"\n\tThis is Display1( ) method of Derived Class";
          }
          void Display2( )
          {
                cout<<"\n\tThis is Display2( ) method of Derived Class";
          }
   };

void main()
```

```
{
        DerivedClass D;

        D.Display1( );      // This will invoke Display1() method of Derived Class
        D.Display2( );      // This will invoke Display2() method of Derived Class
        D.Display3( );      // This will invoke Display3() method of Derived Class
}
```

Output :

```
This is Display1() method of Derived Class
This is Display2() method of Derived Class
This is Display3() method of Base Class
```

VIRTUAL CONSTRUCTORS / DESTRUCTORS

Virtual Destructor:

- The explicit destroying of object with the use of delete operator to a base class pointer to the object is performed by the destructor of the base-class is invoked on that object.
- The above process can be simplified by declaring a virtual base class destructor.
- All the derived class destructors are made virtual in spite of having the same name as the base class destructor.
- In case the object in the hierarchy is destroyed explicitly by using delete operator to the base class pointer to a derived object, the appropriate destructor will be invoked.

Virtual Constructor:

In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.

But virtual destructor is possible.

Example Code

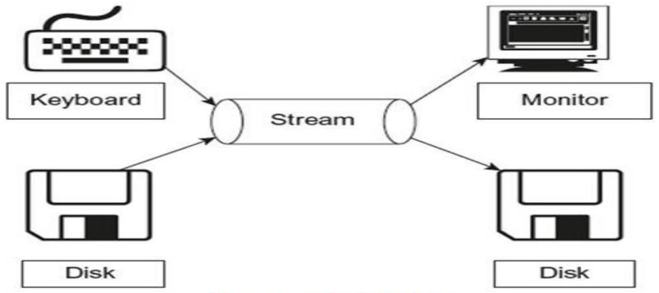
```
#include<iostream>

using namespace std;

class b {

   public:

      b()  {

         cout<<"Constructing base \n";

      }

      virtual ~b()  {

         cout<<"Destructing base \n";

      }

};

class d: public b {

   public:

      d()  {

         cout<<"Constructing derived \n";

      }

      ~d()  {

         cout<<"Destructing derived \n";

      }

};
```

```
int main(void) {

   d *derived = new d();

   b *bptr = derived;

   delete bptr;

   return 0;

}
```

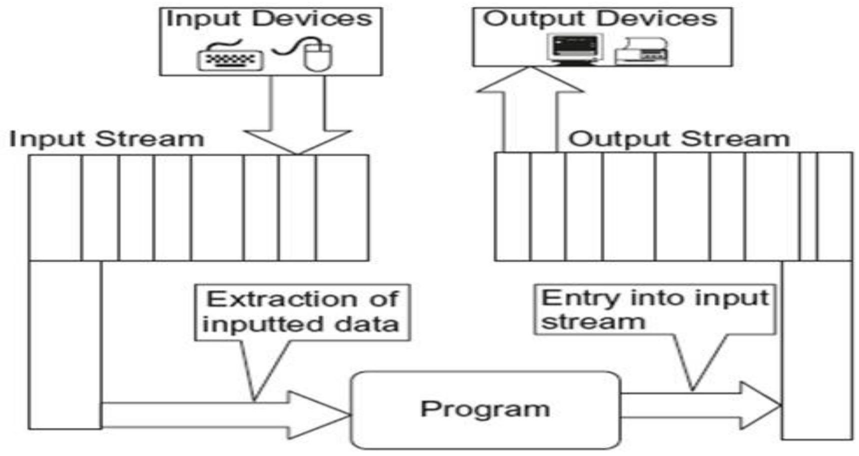
UNIT - IV
MANAGING CONSOLE I/O OPERATIONS

STREAMS:

- A stream is an abstraction. It is a sequence of bytes.
- It represents a device on which input and output operations are performed.
- It can be represented as a source or destination of characters of indefinite length.
- A source from which the input data can be obtained or a destination to which the output data can be sent.
- **Input Stream:** The source stream that provides data to the program is called the input stream.
- **Output Stream:** The destination stream that receives output from the program is called the output stream.



Streams and I/O devices

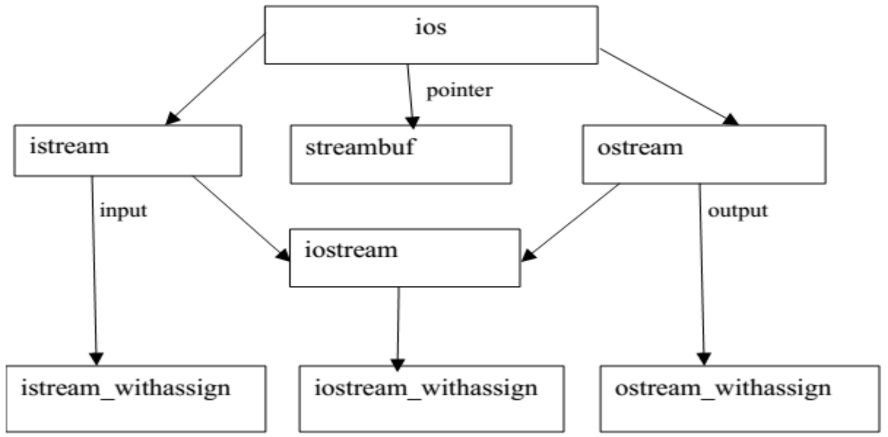


C++ input and output streams

STREAM CLASSES:

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes.Figure below shows the hierarchy of the stream classes used for input and output operations with the console unit.

These classes are declared in the header file iostrem. The file should be included in all programs that communicate with the console unit.



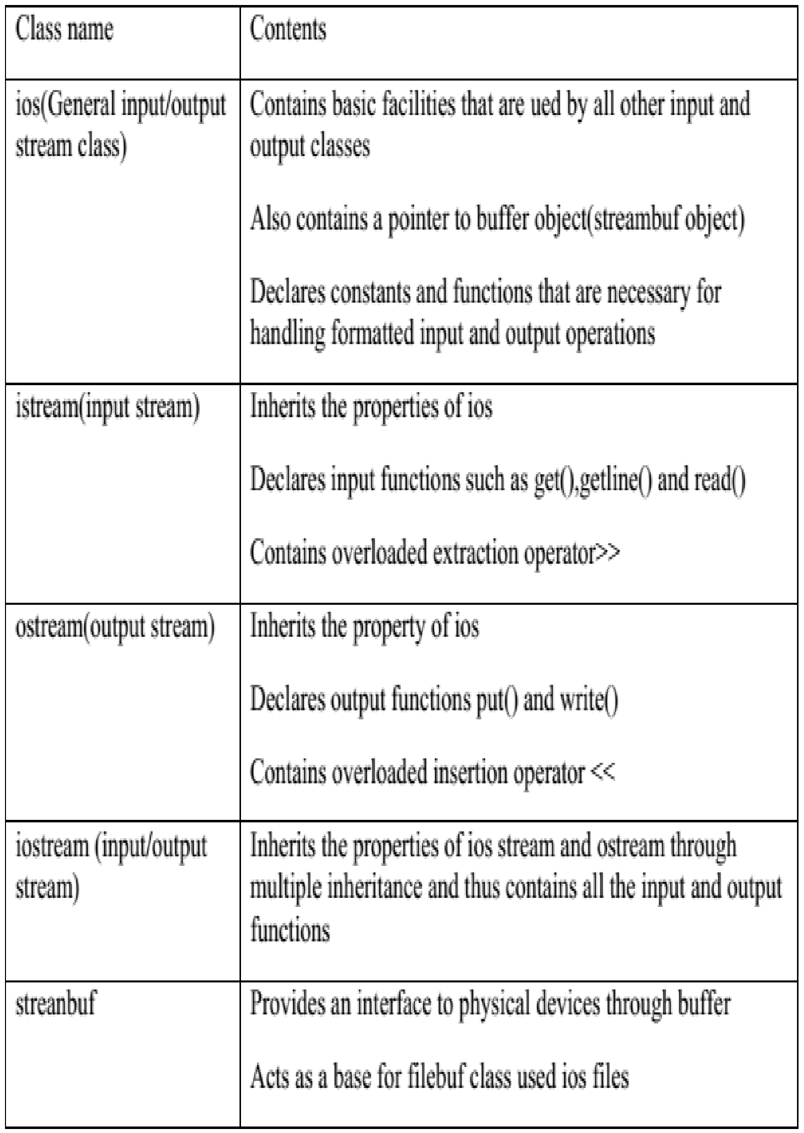
Stream classes for console I/O operations

As in figure above ios is the base class for istream(input stream) and ostream(output stream) which are base classes for iostream(input/output stream).The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted input/output operations.The class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance) provides the facilities for formatted output.

The class iostream provides the facilities for handling both input output streams.Three classes namely istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.

Stream classes for console operations



Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are ued by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streanbuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

There are mainly two types of consol I/O operations form:

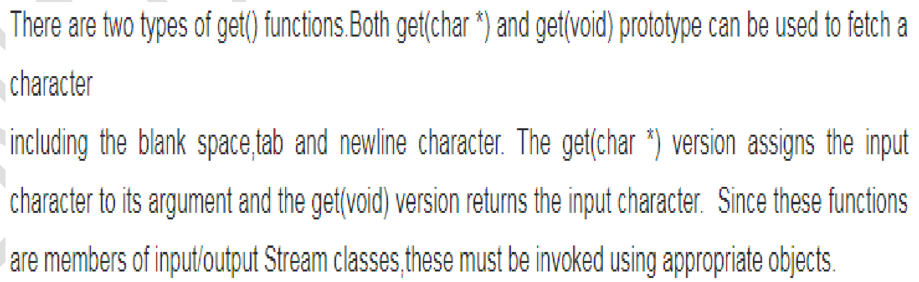
1. Unformatted consol input output
2. Formatted consol input output

UNFORMATTED CONSOL INPUT OUTPUT OPERATIONS

- ➢ These input / output operations are in unformatted mode.
- ➢ The following are operations of unformatted consol input / output operations:

A) get( ) functions:

It is a method of cin object used to input a single character from keyboard. But its main property is that it allows wide spaces and newline character.



There are two types of get() functions.Both get(char *) and get(void) prototype can be used to fetch a character

including the blank space,tab and newline character. The get(char *) version assigns the input character to its argument and the get(void) version returns the input character. Since these functions are members of input/output Stream classes,these must be invoked using appropriate objects.

Syntax:

```
char c=cin.get();
```

Example:

```
#include<iostream>
int main()
{
            char c=cin.get();
            cout<<" C= "<<c<<endl;

            return 0;
}
```

Output:

```
I
C= I
```

B) put( ) Function:

The function **put( )** , a member of **ostream** class, can be used to output a line of text, character by character.

It is a method of cout object and it is used to print the specified character on the screen or monitor.

Syntax:

```
cout.put(variable / character);
```

Example:

```
#include<iostream>
int main()
{
        char c=cin.get();
        cout.put(c); //Here it prints the value of variable c;
        cout.put('c'); //Here it prints the character 'c';

        return 0;
}
```

Output

```
I
Ic
```

C) getline( ) function:

the getline ( ) function reads a whole line of text that ends with a newline characters.

This is a method of cin object and it is used to input a string with multiple spaces.

This function can be invoked by using the object cin as follows:

Syntax:

```
cin.getline ( line, size )
```

This function call invokes the function getline() which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size-1 character are read(whichever occurs first). The newline character is read but not saved. Instead , it is replaced by the null character.

Eg:

```
char x[30];
cin.getline(x,30);
```

Example:

```
#include<iostream>
void main()
{
        cout<<"Enter name :";
        char c[10];
        cin.getline(c,10); //It takes 10 charcters as input;
        cout<<c<<endl;
}
```

Output

```
Enter name :varsha
varsha
```

D) write( ) function:

The write ( ) function displays an entire line .

Syntax:

```
cout.write(line, size)
```

It is a method of cout object. This method is used to read n character from buffer variable.

Eg.

```
cout.write(x,2);
```

Example:

```
#include<iostream>
void main()
{
        cout<<"Enter name : ";
        char c[10];
        cin.getline(c,10); //It takes 10 charcters as input;
        cout.write(c,9); //It reads only 9 character from buffer c;
}
```

Output

```
Enter name : Divyanshux
Divyanshu
```

Input/Output Streams

The **iostream** standard library provides **cin** and **cout** object.
Input stream uses **cin** object for **reading the data** from standard input and Output stream uses **cout**object for **displaying the data** on the screen or writing to standard output.
The **cin** and **cout** are pre-defined streams for input and output data.

Syntax:
cin>>variable_name;
cout<<variable_name;

The **cin** object uses **extraction operator (>>)** before a variable name while the **cout** object uses **insertion operator (<<)** before a variable name.
The cin object is used to read the data through the input device like keyboard etc. while the cout object is used to perform console write operation.

Example: Program demonstrating cin and cout statements

```
#include<iostream>
void  main()
{
        char sname[15];
        cout<<"Enter Employee Name : "<<endl;
        cin>>sname;
        cout<<"Employee Name is : "<<sname;
}
```

Output:

```
Enter Employee Name : Prajakta
Employee Name is : Prajakta
```

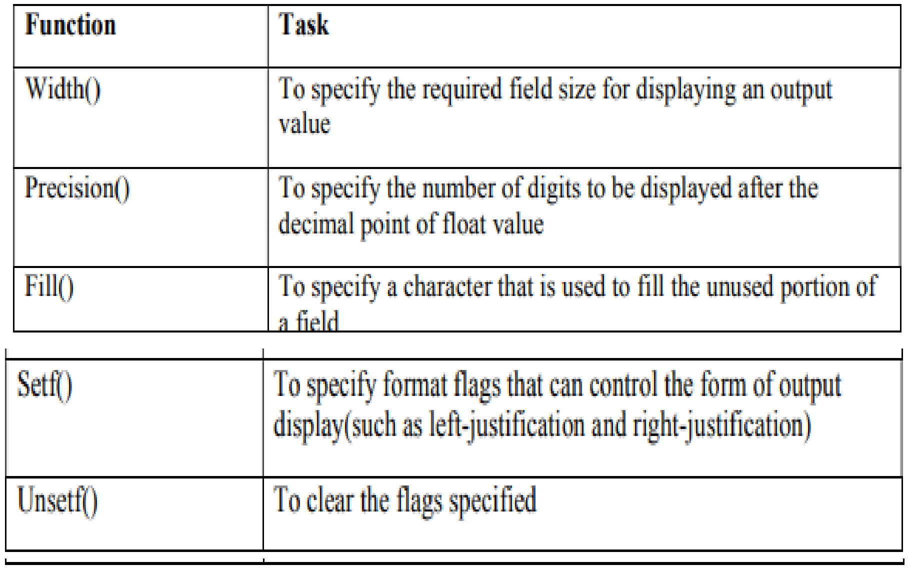
In the above example, **cout<<"Employee Name is : "<<sname** displays the contents of character array **sname** (student name). The cout statement is like printf statement as used in C language.
The cin statement **cin>>sname** reads the string through keyboard and stores in the array **sname[15]**. The cin statement is like scanf statement as used in C language. The **endl** is a manipulator that breaks a line.

2) FORMATTED CONSOLE I/O OPERATIONS

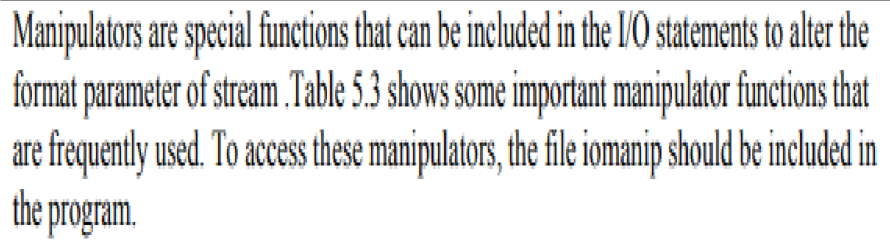
C++ support avoid verity of feature to perform output in different format. They include the following.

1. **ios stream class member function and flags**
2. **Standard manipulator**
3. **User define manipulator**

ios stream class member function and flag: – The stream class, ios contents a large number of member function to assist in formatting the output in a number of ways. The most important among these function are **width(), precision(), fill(), setf() and unsetf().**



Function	Task
Width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of float value
Fill()	To specify a character that is used to fill the unused portion of a field
Setf()	To specify format flags that can control the form of output display(such as left-justification and right-justification)
Unsetf()	To clear the flags specified



Manipulators are special functions that can be included in the I/O statements to alter the format parameter of stream .Table 5.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

Table 5.3 Manipulators

Manipuators	Equivalent ios function
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

In addition to these standard library manipulators we can create our own manipulator functions to provide any special output formats.

A) Defining Field Width: width( )

This function is used to set width of the output.

Syntax:

```
cout.width(w);
```

here w is the field width.The output will be printed in a field of w character wide at the right end of field.The width() function can specify the field width for only one item(the item that follows immediately).After printing one item(as per the specification) it will revert back the default.for example,the statements

```
cout.width(5);
cout<<543<<12<<"\n";
```

will produce the following output:

		5	4	3	1	2

The value 543 is printed right justified in the first five columns.The specification width(5) does not retain the setting for printing the number 12.this can be improved as follows:

```
cout.width(5);
cout<<543;
cout.width(5);
cout<<12<<"\n";
```

This produces the following output:

		5	4	3				1	2

B) Setting Precision: precision( )

By default ,the floating numbers are printed with six digits after the decimal points. However ,we can specify the number of digits to be displayed after the decimal point while printing the floating point numbers.

This can be done by using the precision () member function as follows:

```
cout.precision(d);
```

where d is the number of digits to the right of decimal point.for example the statements

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
cout<<3.14159<<"\n";
cout<<2.50032<<"\n";
```

will produce the following output:

```
1.141      (truncated)
3.142      (rounded to nearest cent)
2.5        (no trailing zeros)
```

Unlike the function width(),precision() retains the setting in effect until it is reset.That is why we have declared only one statement for precision setting which is used by all the three outputs.We can set different valus to different precision as follows:

```
cout.precision(3);

cout<<sqrt(2)<<"\n";

cout.precision(5);

cout<<3.14159<<"\n";
```

We can also combine the field specification with the precision setting.example:

```
cout.precision(2);

cout.width(5);

cout<<1.2345;
```

The first two statement instruct :"print two digits after the decimal point in a field of five character width".Thus the output will be:

	1		2	3

[illegible]

C) Filling and Padding : fill( )

The unused portion of field width are filled with white spaces, by default. The fill( ) function can be used to fill the unused positions by any desired character.

Syntax:

cout.fill (ch);

where ch represents the character which is used for filling the unused positions.

cout.fill ('*')
cout.width(10);
cout<<5250<<"\n";

The output would be:

*	*	*	*	*	*	5	2	5	0

D) Formatting Flags, Bit-fields and setf( ):

The setf() a member function of the ios class, can provide answers left justified.The setf() function can be used as follows:

cout.setf(arg1.arg2)

The arg1 is one of the formatting flags defined in the class ios.The formatting flag specifies the format action required for the output.Another ios constant,arg2,known as bit field specifies the group to which the formatting flag belongs. for example:

```
cout.setf(ios::left,ios::adjustfield);

cout.setf(ios::scientific,ios::floatfield);
```

Note that the first argument should be one of the group member of second argument.

Consider the following segment of code:

```
cout.fill('*');

cout.setf(ios::left,ios::adjustfield);

cout.width(15);

cout<<"table1"<<"\n";
```

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
cout.fill('*');

cout.precision(3);

 cout.setf(ios::internal,ios::adjustfield);

 cout.setf(ios::scientific,ios::floatfield);

 cout.width(15);

 cout<<-12.34567<<"\n";
```

Will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

What is a Manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.

The header file **iomanip** provides a set of functions called manipulators which can be used to manipulate the output formats.

#include<iomanip>

There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

This manipulator has the same functionality as the 'n' newline character.

For example:

```
1.          cout << "Exforsys" << endl;
2.          cout << "Training";
```

Output:

```
Exforsys
Training
```

setw Manipulator:

This manipulator sets the minimum field width on output.

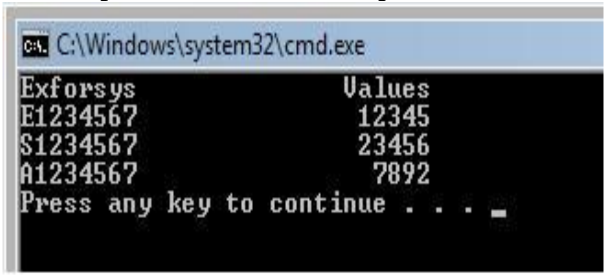
The syntax is:

setw(x)

Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is .

```
 #include <iostream>
#include <iomanip>
void main( )
{
int x1=12345,x2= 23456, x3=7892;
cout << setw(8) << "Exforsys" << setw(20) << "Values" <<
endl
 << setw(8) << "E1234567" << setw(20)<< x1 << endl
 << setw(8) << "S1234567" << setw(20)<< x2 << endl
 << setw(8) << "A1234567" << setw(20)<< x3 << endl;
}
```

The output of the above example is:



C:\Windows\system32\cmd.exe

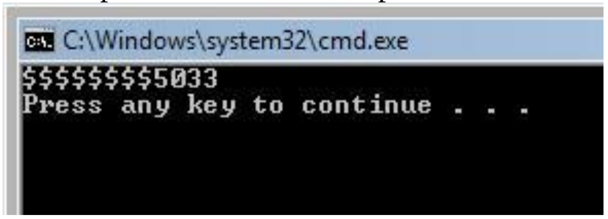
```
Exforsys                   Values
E1234567                    12345
S1234567                    23456
A1234567                     7892
Press any key to continue . . .
```

setfill Manipulator:

This is used after setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

```
 #include <iostream>
#include <iomanip>
void main()
{
cout << setw(10) << setfill('$') << 50 << 33 << endl;
}
```

The output of the above example is:



C:\Windows\system32\cmd.exe

```
$$$$$$$$5033
Press any key to continue . . .
```

This is because the setw sets 10 for the width of the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

setprecision Manipulator:

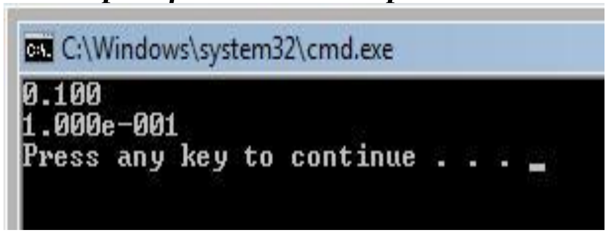
The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

- fixed
- scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator. The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation. The keyword scientific, before the setprecision manipulator, prints the floating point number in scientific notation.

```
 #include <iostream>
#include <iomanip>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << scientific << x << endl;
}
```

The output of the above example is:



```
C:\Windows\system32\cmd.exe
0.100
1.000e-001
Press any key to continue . . .
```

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

<u>**WORKING WITH FILES**</u>

In C++, you open a file, you must first obtain a stream. There are the following three types of streams:

- input
- output
- input/output

Create an Input Stream

To create an input stream, you must declare the stream to be of class ifstream. Here is the syntax:

ifstream fin;

Create an Output Stream

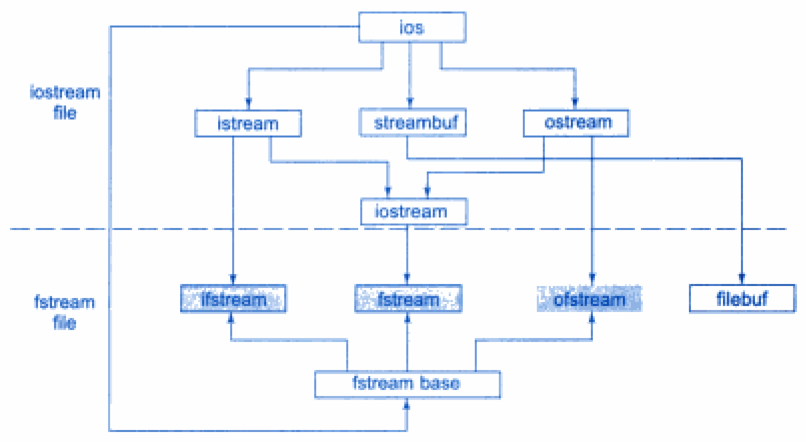
To create an output stream, you must declare it as class ofstream. Here is an example:

ofstream fout;

Create both Input/Output Streams

Streams that will be performing both input and output operations must be declared as class fstream. Here is an example:

fstream fio;



Stream classes for file operations

Opening a File in C++

Once a stream has been created, next step is to associate a file with it. And thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways :

1. Using the constructor function of the stream class.
2. Using the function open().

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let's discuss each of these methods one by one.

A) Opening File Using Constructors

We know that a constructor of class initializes an object of its class when it (the object) is being created. Same way, the constructors of

stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them. This is carried out as explained here:
To open a file named myfile as an input file (i.e., data will be need from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type.
Here is an example:

```
ifstream fin("myfile", ios::in) ;
```

The above given statement creates an object, fin, of input file stream. The object name is a user-defined name (i.e., any valid identifier name can be given). After creating the ifstream object fin, the file myfile is opened and attached to the input stream, fin. Now, both the data being read from myfile has been channelised through the input stream object.
Now to read from this file, this stream object will be used using the getfrom operator (">>").
Here is an example:

```
char ch;
fin >> ch ;     // read a character from the file
float amt ;
fin >> amt ;    // read a floating-point number form the file
```

Similarly, when you want a program to write a file i.e., to open an output file (on which no operation can take place except writing only). This will be accomplish by

1. creating ofstream object to manage the output stream
2. associating that object with a particular file

Here is an example,

```
ofstream fout("secret" ios::out) ;     // create ofstream object named as fout
```

This would create an output stream, object named as fout and attach the file secret with it.
Now, to write something to it, you can use << (put to operator) in familiar way. Here is an example,

```
int code = 2193 ;
fout << code << "xyz" ;   /* will write value of code
                             and "xyz" to fout's associated
                             file namely "secret" here. */
```

The connections with a file are closed automatically when the input and the output stream objects expires i.e., when they go out of scope. (For example, a global object expires when the program terminates). Also, you can close a connection with a file explicitly by using the close() method :

```
fin.close() ;     // close input connection to file
fout.close() ;    // close output connection to file
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. The stream still remains there. For example, after the above statements, the streams fin and fout still exist along with the buffers they manage. You can reconnect the stream to the same file or to another file, if required. Closing a file flushes the buffer which means the data remaining in the buffer (input or output stream) is moved out of it in the direction it is ought to be.
For example, when an input file's connection is closed, the data is moved from the input buffer to the program and when an output file's connection is closed, the data is moved from the output buffer to the disk file.

B) Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e., processing them one by one), then you can open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file.

For example,

```
ifstream fin;                      // create an input stream
fin.open("Master.dat", ios::in);   // associate fin stream with file Master.dat
:                                  // process Master.dat
fin.close();                       // terminate association with Master.dat
```

```
        fin.open("Tran.dat", ios::in);      // associate fin stream with file
        Tran.dat
        :                                   // process Tran.dat
        fin.close();                        // terminate association
```

The above code lets you handle reading two files in succession. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time.

C) The Concept of File Modes

The file mode describes how a file is to be used : to read from it, to write to it, to append it, and so on.

When you associate a stream with a file, either by initializing a file stream object with a file name or by using the open() method, you can provide a second argument specifying the file mode, as mentioned below :

stream_object.open("filename", (filemode)) ;

The second method argument of open(), the filemode, is of type int, and you can choose one from several constants defined in the ios class.

List of File Modes in C++

Following table lists the filemodes available in C++ with their meaning :

Constant	Meaning	Stream Type
ios :: in	It opens file for reading, i.e., in input mode.	ifstream
ios :: out	It opens file for writing, i.e., in output mode. This also opens the file in ios :: trunc mode, by default. This means an existing file is truncated when opened, i.e., its previous contents are discarded.	ofstream
ios :: ate	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	ofstream ifstream
ios :: app	This causes all output to that file to be appended to the end. This value can be used only with files capable of output.	ofstream
ios :: trunc	This value causes the contents of a pre-existing file by the same name to be destroyed and truncates the file to zero length.	ofstream
ios :: nocreate	This cause the open() function to fail if the file does not already exist. It will not create a new file with that name.	ofstream
ios :: noreplace	This causes the open() function to fail if the file already exists. This is used when you want to create a new file and at the same time.	ofstream
ios :: binary	This causes a file to be opened in binary mode. By default, files are opened in text mode. When a file is opened in text mode, various character translations may take place, such as the conversion of carriage-return into newlines. However, no such character translations occur in file opened in binary mode.	ofstream ifstream

If the ifstream and ofstream constructors and the open() methods take two arguments each, how have we got by using just one in the previous examples ? As you probably have guessed, the prototypes for these class member functions provide default values for the second argument (the filemode argument). For example, the ifstream open()

method and constructor use ios :: in (open for reading) as the default value for the mode argument, while the ofstream open() method and constructor use ios :: out (open for writing) as the default.
The fstream class does not provide a mode by default and, therefore, one must specify the mode explicitly when using an object of fstream class.
Both ios::ate and ios::app place you at the end of the file just opened. The difference between the two is that the ios::app mode allows you to add data to the end of the file only, when the ios::ate mode lets you write data anywhere in the file, even over old data.
You can combine two or more filemode constants using the C++ bitwise OR operator (symbol |).

For example, the following statement :

```
ofstream fout;
fout.open("Master", ios :: app | ios :: nocreate);
```

will open a file in the append mode if the file exists and will abandon the file opening operation if the file does not exist.
To open a binary file, you need to specify ios :: binary along with the file mode, e.g.,

```
fout.open("Master", ios :: app | ios :: binary);
```

or,

```
fout.open("Main", ios :: out | ios :: nocreate | ios :: binary);
```

Closing a File in C++

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. The close() function accomplishes this task and it takes the following general form :

```
stream_object.close();
```

For example, if a file Master is connected with an ofstream object fout, its connections with the stream fout can be terminated by the following statement :

```
fout.close() ;
```

C++ Opening and Closing a File Example

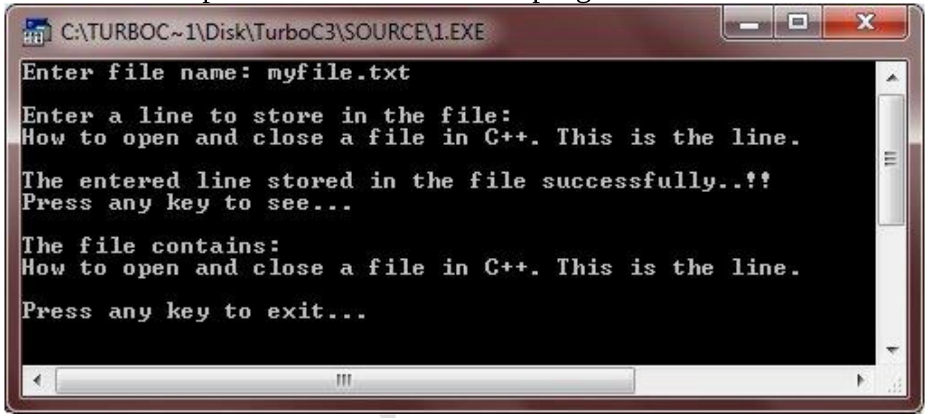
Here is an example given, for the complete understanding on:

- how to open a file in C++ ?
- how to close a file in C++ ?

Let's look at this program.

```
/* C++ Opening and Closing a File
 * This program demonstrates, how
 * to open a file to store or retrieve
 * information to/from it. And then how
 * to close that file after storing
 * or retrieving the information to/from it. */

#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
        ofstream fout;
        ifstream fin;
        char fname[20];
        char rec[80], ch;
        clrscr();

        cout<<"Enter file name: ";
        cin.get(fname, 20);

        fout.open(fname, ios::out);

        if(!fout)
        {
                cout<<"Error in opening the file "<<fname;
                getch();
                exit(1);
        }
        cin.get(ch);

        cout<<"\nEnter a line to store in the file:\n";
        cin.get(rec, 80);
        fout<<rec<<"\n";
        cout<<"\nThe entered line stored in the file successfully..!!";
        cout<<"\nPress any key to see...\n";
        getch();
        fout.close();

        fin.open(fname, ios::in);
        if(!fin)
        {
                cout<<"Error in opening the file "<<fname;
                cout<<"\nPress any key to exit...";
                getch();
                exit(2);
        }

        cin.get(ch);
        fin.get(rec, 80);
        cout<<"\nThe file contains:\n";
        cout<<rec;
        cout<<"\n\nPress any key to exit...\n";
        fin.close();

        getch();
}
```

Here is the sample run of the above C++ program:



```
Enter file name: myfile.txt

Enter a line to store in the file:
How to open and close a file in C++. This is the line.

The entered line stored in the file successfully..!!
Press any key to see...

The file contains:
How to open and close a file in C++. This is the line.

Press any key to exit...
```

File Pointers And Manipulators

All file objects hold two file pointers that are associated with the file. These two file pointers provide two integer values. These integer values indicate the exact position of the file pointers in the number of bytes in the file. The read or write operations are carried out at the location pointed by these file pointers .One of them is called get pointer (input pointer), and the second one is called put pointer (output pointer). During reading and writing operations with files, these file pointers are shifted from one location to another in the file. The (input) get pointer helps in reading the file from the given location, and the output pointer helps in writing data in the file at the specified location. When read and write operations are carried out, the respective pointer is moved.

While a file is opened for the reading or writing operation, the respective file pointer input or output is by default set at the beginning of the file. This makes it possible to perform the reading or writing operation from the beginning of the file. The programmer need not explicitly set the file pointers at the beginning of files. To explicitly set the file pointer at the specified position, the file stream classes provides the following functions:

Read mode: When a file is opened in read mode, the get pointer is set at the beginning of the file, as shown in Figure. Hence, it is possible to read the file from the first character of the file.

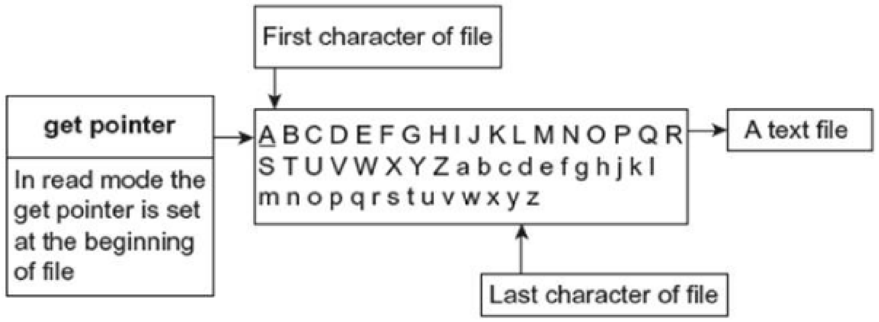


Fig: Status of get pointer in read mode

Write Mode: When a file is opened in write mode, the put pointer is set at the beginning of the file, as shown in Figure. Thus, it allows the write operation from the beginning of the file. In case the specified file already exists, its contents will be deleted.

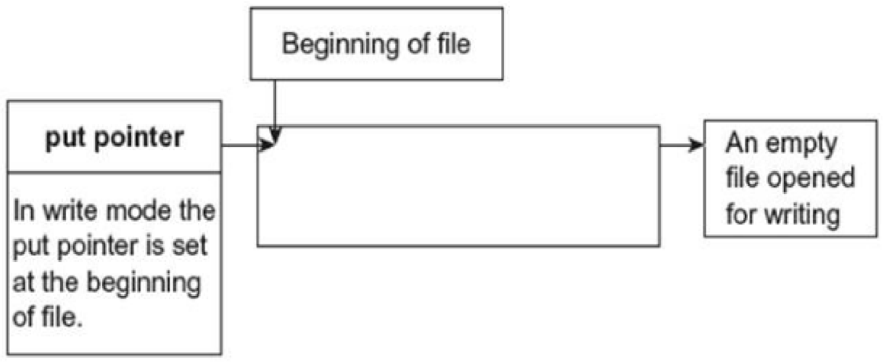

Beginning of file
put pointer
In write mode the put pointer is set at the beginning of file.
An empty file opened for writing

Fig: Status of put pointer in write mode

Append Mode: This mode allows the addition of data at the end of the file. When the file is opened in append mode, the output pointer is set at the end of the file, as shown in Figure. Hence, it is possible to write data at the end of the file. In case the specified file already exists, a new file is created, and the output is set at the beginning of the file. When a pre-existing file is successfully opened in append mode, its contents remain safe and new data are appended at the end of the file.

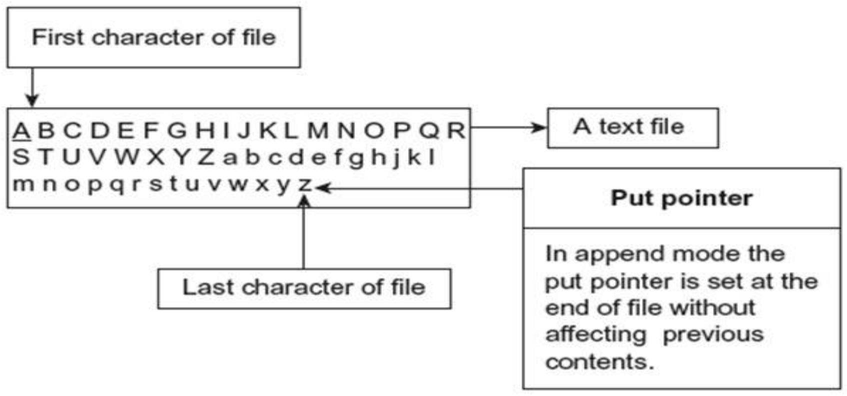

First character of file
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h j k l m n o p q r s t u v w x y z
A text file
Put pointer
In append mode the put pointer is set at the end of file without affecting previous contents.
Last character of file

Fig: Status of put pointer in append mode

C++ has four functions for the setting of points during file operation. The position of the curser in the file can be changed using these functions. These functions are described in Table.

Table: File pointer handling functions

Function	**Uses**	**Remark**
seekg()	Shifts input (get) pointer to a given location.	Member of ifstream class
seekp()	Shifts output (put) pointer to a given location.	Member of ofstream class
tellg()	Provides the present position of the input pointer.	Member of ifstream class
tellp()	Provides the present position of the output pointer.	Member of ofstream class

As given in Table, the seekg() and tellg() are member functions of the ifstream class. All the above four functions are present in the class fstream. The class fstream is derived from ifstream and ofstream classes. Hence, this class supports both input and output modes, as shown in Figure. The seekp() and tellp() work with the put pointer, and tellg() and seekg() work with the get pointer.

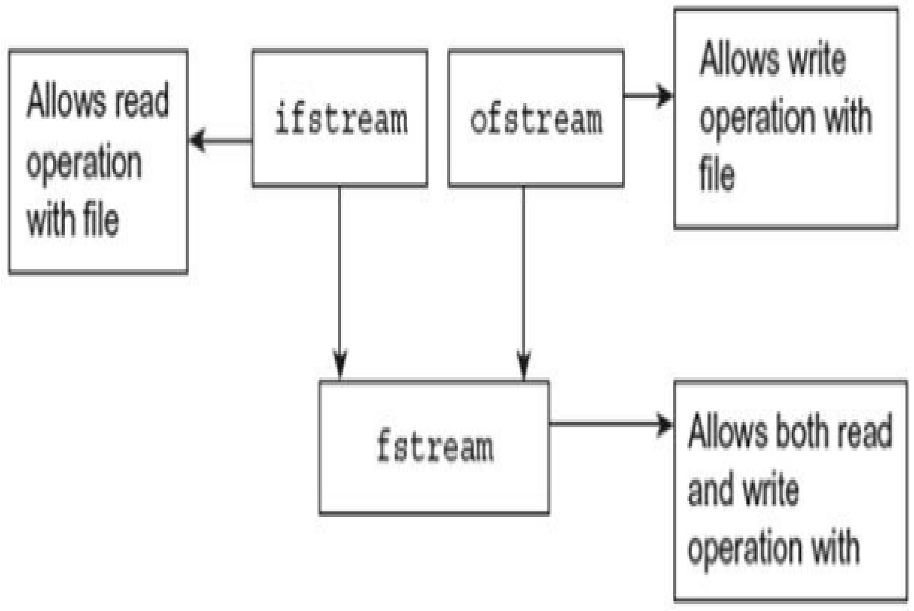

Allows read operation with file
ifstream
ofstream
Allows write operation with file
fstream
Allows both read and write operation with

Fig: Derivation of fstream class

16.11 Write a program to append a file.

```
#include<fstream.h>
#include<conio.h>
int main()
{
    clrscr();
    ofstream out;
    char data[25];
    out.open ("text",ios::out);
    cout<<"\n Enter text"<<endl;
    cin.getline(data,25);
    out <<data;
    out.close();
    out.open ("text", ios::app );
    cout<<"\n Again Enter text"<<endl;
    cin.getline (data,25);
    out<<data;
    out.close();
    ifstream in;
    in.open("text", ios::in);
    cout<<endl<<"Contents of the file \n";
    while (in.eof()==0)
    {
    in>>data;
    cout<<data;
    }
    return 0;
 }
```

```
OUTPUT
Enter text
C-PLUS-
Again Enter text
PLUS
Contents of the file
C-PLUS-PLUS
```

16.12 Write a program to read contents of the file. Display the position of the get pointer.

```
#include<fstream.h>
#include<conio.h>
int main()
{
   clrscr();
   ofstream out;
   char data[32];
   out.open ("text",ios::out);
   cout<<"\n Enter text"<<endl;
   cin.getline(data,32);
   out <<data;
   out.close();
   ifstream in;
   in.open("text", ios::in);
   cout<<endl<<"Contents of the file \n";
   int r;
    while (in.eof()==0)
    {
    in>>data;
    cout<<data;
    r=in.tellg();
    cout<<" ("<<r <<")";
    }
    return 0;
 }
```

```
OUTPUT
Enter text
Programming In ANSI and TURBO-C
Contents of the file
Programming (11)In (14)ANSI (19)and (23)TURBO-C (31)
```

Sequential Access Files

C++ allows the file manipulation command to access the file sequentially or randomly. The data of the sequential file should be accessed sequentially, that is, one character at a time. In order to access the nth number of bytes, all previous characters are read and ignored. There are a number of functions to perform read and write operations with the files. Some functions read /write single characters, and some functions read/write blocks of binary data. The put() and get() functions are used to read or write a single character, whereas write() and read() are used to read or write blocks of binary data.

put() and get() functions

The function get() is a member function of the class fstream. This function reads a single character from the file pointed by the get pointer, that is, the character at the current get pointer position is caught by the get() function.

The function put() function writes a character to the specified file by the stream object. It is also a member of the fstream class. The put() function places a character in the file indicated by the put pointer.

16.15 Write a program to write and read string to the file using put() and get() functions.

```
#include<fstream.h>
#include<conio.h>
#include<string.h>
int main()
{
   clrscr();
   char text[50];
   cout<<"\n Enter a Text:";
   cin.getline(text,50);
   int l=0;
   fstream io;
   io.open("data", ios::in | ios::out);
   while (l[text]!='\0')
   io.put(text[l++]);
   io.seekg(0);
   char c;
   cout<<"\n Entered Text:";
   while (io)
   {
   io.get(c);
   cout<<c;
   }
   return 0;
}
```

OUTPUT

```
Enter a Text : PROGRAMMING WITH C++
Entered Text : PROGRAMMING WITH C++
```

Random Access Operation

Data files always contain a large amount of information, and the information always changes. The changed information should be updated; otherwise, the data files are not useful. Thus, to update data in the file, we need to update the data files with latest information. To update a particular record of the data file, the data may be stored anywhere in the file; it is necessary to obtain the location (in terms of byte number) at which the data object is stored.

The sizeof() operator determines the size of the object. Consider the following statements:

```
(a) int size = sizeof(o);
```

Here, o is an object, and size is an integer variable. The sizeof() operator returns the size of the object o in bytes, and it is stored in the variable size. Here, one object is equal to one record.

The position of the nth record or object can be obtained using the following statement:

```
(b) int p = (n-1 * size);
```

Here, p is the exact byte number of the object that is to be updated; n is the number of the object; and size is the size in bytes of an individual object (record).

Suppose we want to update the fifth record. The size of the individual object is 26.

```
(c) p = (5-1*26) i.e. p = 104
```

Thus, the fifth object is stored in a series of bytes from 105 to 130. Using seekg() and seekp() functions, we can set the file pointer at that position.

16.19 Write a program to create a text file. Add and modify records in the text file. The record should contain name, age, and height of a boy.

```
#include<stdio.h>
#include<process.h>
#include<fstream.h>
#include<conio.h>
class boys
{
   char name [20];
   int age;
   float height;
   public:
   void input()
   {
   cout<< "Name:"; cin>>name;
   cout<< "Age:"; cin>>age;
   cout<< "Height:"; cin>>height;
   }
   void show (int r)
   {
   cout<<"\n"<<r<<"\t"<<name<<"\t"<<age <<"\t"<<height; }
};
boys b[3];
fstream out;
void main()
{
clrscr();
void menu (void);
out.open ("boys.doc", ios::in | ios::out | ios::noreplace);
menu();
}
void menu(void)
{
void get(void);
void put(void);
void update(void);
int x;
clrscr();
cout<<"\n Use UP arrow key for selection";
char ch=' ';
gotoxy(1,3);
printf ("ADD()");
gotoxy(1,4);
printf ("ALTER()");
gotoxy(1,5);
printf ("EXIT()");
x=3;
gotoxy(7,x);
printf ("*");
while (ch!=13)
{
ch=getch();
if (ch==72)
{
if (x>4)
{
gotoxy(7,x);
printf (" ");
x=2;
}
gotoxy(7,x);
printf (" ");
gotoxy(7,++x);
printf ("*");
}
}
switch(x)
{
case 3 : get(); put(); getche(); break;
case 4 : put(); update(); put(); getche(); break;
default : exit(1);
}
menu();
}
void get()
{
cout<<"\n\n\n\n Enter following information:\n";
for (int i=0;i<3;i++)
{
b[i].input();
out.write ((char*) & b[i],sizeof(b[i]));
}
}
```

```
void put()
{
out.seekg(0,ios::beg);
cout<<"\n\n\n Entered information \n";
cout<<"Sr.no Name Age Height";
for (int i=0;i<3;i++)
{
out.read((char *) & b[i],
sizeof(b[i]));
b[i].show(i+1);
}
}
void update()
{
int r, s=sizeof(b[0]);
out.seekg(0,ios::beg);
cout<<"\n"<<"Enter record no. to update:";
cin>>r;
r=(r-1)*s;
out.seekg(r,ios::beg);
b[0].input();
out.write ((char*) & b[0],sizeof(b[0]));
put();
}
```

OUTPUT

```
Use UP arrow key for selection
ADD (*)
ALTER()
EXIT()
Enter following information :
Name : Sachin
Age : 28
Height : 5.4
Name : Rahul
Age : 28
Height : 5.5
Name : SauravAge : 29
Height : 5.4
Entered information
Sr.no Name Age Height
1 Sachin 28 5.4
2 Rahul 28 5.5
3 Saurav 29 5.4
```

Error Handling Functions

Until now, we have performed the file operation without any knowledge of the failure or success of the function open() that opens the file. There are many reasons; they may result in errors during read/write operations of the program.

1. An attempt to read a file that does not exist
2. The file name specified for opening a new file may already exist
3. An attempt to read the contents of a file when the file pointer is at the end of the file
4. Insufficient disk space
5. Invalid file name specified by the programmer
6. An effort to write data to the file that is opened in the read-only mode
7. A file opened may be already opened by another program
8. An attempt to open the read-only file for writing operation
9. Device error

The stream state member from the class ios receives values from the status bit of the active file. The class ios also contains many different member functions. These functions read the status bit of the file where an error occurred during program execution are stored. These functions are depicted in Table, and various status bits are described in Table.

All streams such as ofstream, ifstream, and fstream contain the state connected with them. Faults and illegal conditions are managed (controlled) by setting and checking the state properly. Figure describes it more clearly.

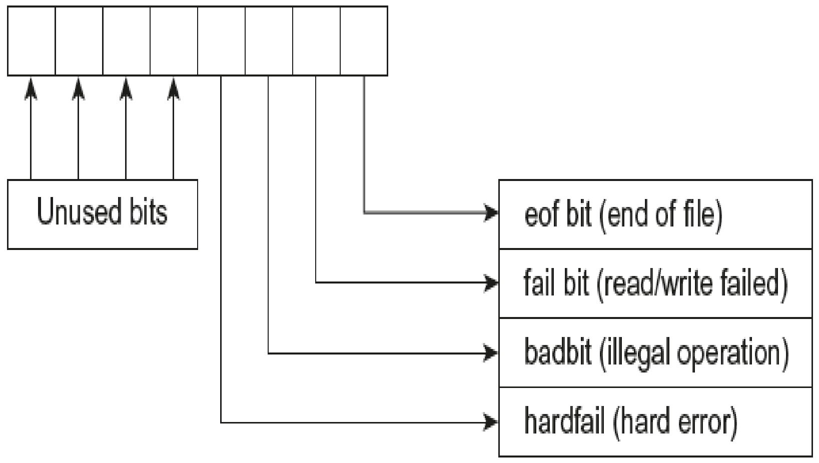


Fig: Status bits

Table: Status Bits

eofbit	End of file encountered.	0x01
failbit	Operation unsuccessful	0x02
badbit	Illegal operation due to wrong size of buffer	0x04
hardfail	Critical error	0x08

Table: Error trapping functions

Functions	**Working and return value**
fail()	Returns non-zero value if an operation is unsuccessful. This is carried out by reading the bits ios::failbit, ios:: badbit, and ios::hardfail of ios::state.
eof()	Returns non-zero value when the end of the file is detected; otherwise, it returns zero. The ios::eofbit is checked.
bad()	Returns non-zero value when an error is found in the operation. The ios::badbit is checked.
good()	Returns non-zero value if no error occurred during the file operation, that is, no status bits were set. This also indicates that the above functions are false. When this function returns true, we can proceed with the file operation.
rdstate()	Returns the stream state. It returns the value of various bits of the ios::state.

The following examples illustrate the techniques of error checking:

a. **An attempt to open a non-existent file for reading**

```
ifstream in(data.txt");
if (!in){ cout<< File not found"; }
```

b. In the above format, an attempt is made to open a file for r the file already exists, it will be opened; otherwise, the ope fails. Thus, by checking the value of the object in, we can c the failure or success of the operation and according to th further processing can be decided.

c. **An attempt to open a read-only file for writing**

```
ofstream out(data.txt");
if (!out)
cout<<Unable to open file";
else
cout<<"File opened";
```

d. Suppose the data.txt file is protected (marked read only) or used by another application in a multitasking operating environment. If the same file is opened in the write mode as shown above, the operation fails. By checking the value of the object out with the if() statement, we can catch the error and transfer the program control to a suitable sub-routine.

e. **Checking end of file**

```
ifstream in(data.txt");
while (!in.eof())
{
// read data from file
// display on screen
}
```

Command-Line Arguments

An executable program that performs a specific task for the operating system is called a command. The commands are issued from the command prompt of the operating system. Some arguments are associated with the commands; hence these arguments are called command-line arguments. These associated arguments are passed to programs.

Similar to C, in C++, every program starts with a main() function, and this function marks the beginning of the program. We have not provided any arguments so far in the main() function. Here, we can make arguments in the main function as in other functions. The main()function can receive two arguments, and they are (1) argc (argument counter) and (2) argv (argument vector). The first argument contains the number of arguments, and the second argument is an array of char pointers. The *argv points to the command-line arguments. The size of the array is equal to the value counted by the argc. The information contained in the command line is passed on to the program through these arguments when the main() is called up by the system.

1. **Argument argc:** The argument argc counts the total number of arguments passed from command prompt. It returns a value that is equal to the total number of arguments passed through the main().
2. **Argument argv:** It is a pointer to an array of character strings that contains names of arguments. Each word is an argument.

```
Syntax - main ( int argc, char * argv[]);
Example - ren file1 file2.
```

Here, file1 and file2 are arguments, and copy is a command. The first argument is always an executable program followed by associated arguments. If you do not specify the argument, the first program name itself is an argument but the program will not run properly and will flag an error. The contents of argv[] would be as follows:

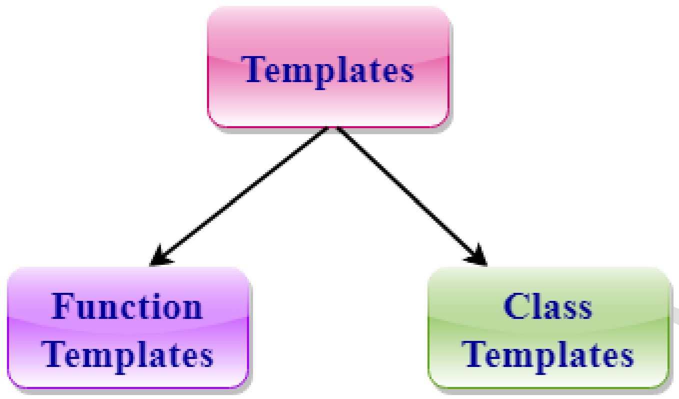
```
argv [0] → ren
argv [1] → file1
argv [2] → file2
```

C++ TEMPLATES

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:
We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:
We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

FUNCTION TEMPLATE

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template:

```
template < class Ttype> ret_type f
unc_name(parameter_list)
{
   // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
 #include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;

}
```

```
int main()
{
 int i =2;
 int j =3;
 float m = 2.3;
 float n = 1.2;
 cout<<"Addition of i and j is :"<<add(i,j);
 cout<<'\n';
 cout<<"Addition of m and n is :"<<add(m,n);
 return 0;
}
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

FUNCTION TEMPLATES WITH MULTIPLE PARAMETERS

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
 template<class T1, class T2,.....>

return_type function_name (argu
ments of type T1, T2....)
{
   // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```
#include <iostream>
template<class X,class Y> void fun(X a,Y b)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
  fun(15,12.3);

  return 0;
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

OVERLOADING A FUNCTION TEMPLATE

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
   std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
   std::cout << "Value of b is : " <<b<< std::endl;
   std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
  fun(10);
  fun(20,30.5);
  return 0;
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

RESTRICTIONS OF GENERIC FUNCTIONS

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}

void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   }

}
int main()
{
  fun(4.6);
  fun(6);
  return 0;
}
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE

- **Class Template** can also be defined similarly to the Function Template.
- When a class uses the concept of Template, then the class is known as generic class.

Syntax:

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

class_name<type> ob;

where class_name: It is the name of the class.
type: It is the type of the data that the class is operating on.
ob: It is the name of the object.
Let's see a simple example:

```
#include <iostream>

template<class T>
class A
{
   public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      std::cout << "Addition of num1 and num2 : " << num1+num2
<<std::endl;
   }

};

int main()
{
   A<int> d;
   d.add();
   return 0;
}
```

Output:

```
Addition of num1 and num2 : 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
  // Body of the class.
}
```

Let's see a simple example when class template contains two generic data types.

```
#include <iostream>

    template<class T1, class T2>
   class A
   {
       T1 a;
       T2 b;
       public:
      A(T1 x,T2 y)    {
        a = x;
        b = y;
      }
        void display()
       {
            std::cout << "Values of a and b are : " << a<<" ,"<<b<<
std::endl;
        }
     };
     void  main()
    {
        A<int,float> d(5,6.5);
        d.display();
    }
```

Output:

```
Values of a and b are : 5,6.5
```

NONTYPE TEMPLATE ARGUMENTS

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

Let' s see the following example:

```
template<class T, int size>
class array
{
      T arr[size];          // automatic array i
nitialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;                    // array of 15 integers.
array<float, 10> t2;                 // array of 10 floats.
array<char, 4> t3;                   // array of 4 chars.
```

Let's see a simple example of nontype template arguments.

```
#include <iostream>
template<class T, int size>
class A
{
   public:
   T arr[size];
   void insert()
   {
      int i =1;
      for (int j=0;j<size;j++)
      {
         arr[j] = i;
         i++;
      }
   }
    void display()
   {
      for(int i=0;i<size;i++)
      {
         std::cout << arr[i] << " ";
      }
   }
};
int main()
{
   A<int,10> t1;
   t1.insert();
   t1.display();
   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

EXCEPTION HANDLING

Exceptions allow a method to react to exceptional circumstances and errors (like runtime errors) within programs by transferring control to special functions called handlers. For catching exceptions, a portion of code is placed under exception inspection. Exception handling was not a part of the original C++. It is a new feature that ANSI C++ included in it. Now almost all C++ compilers support this feature. Exception handling technology offers a securely integrated approach to avoid the unusual predictable problems that arise while executing a program.

There are two types of exceptions:

1. Synchronous exceptions
2. Asynchronous exceptions

Errors such as: out of range index and overflow fall under the category of *synchronous* type exceptions. Those errors that are caused by events beyond the control of the program are called *asynchronous* exceptions. The main motive of the exceptional handling concept is to provide a means to detect and report an exception so that appropriate action can be taken. This mechanism needs a separate error handling code that performs the following tasks:

- Find and hit the problem (exception)
- Inform that the error has occurred (throw exception)
- Receive the error information (Catch the exception)
- Take corrective actions (handle exception)

Mechanism of Exception Handling

The error handling mechanism basically consists of two parts. These are:

1. To detect errors
2. To throw exceptions and then take appropriate actions

Exception handling in C++ is built on three keywords: *try*, *catch*, and *throw*.

- try
- throw: A program throws an exception when a problem is detected which is done using a keyword "throw".
- catch: A program catches an exception with an exception handler where programmers want to handle the anomaly. The keyword catch is used for catching exceptions.

The Catch blocks catching exceptions must immediately follow the try block that throws an exception.

The general form of these two blocks is as follows:

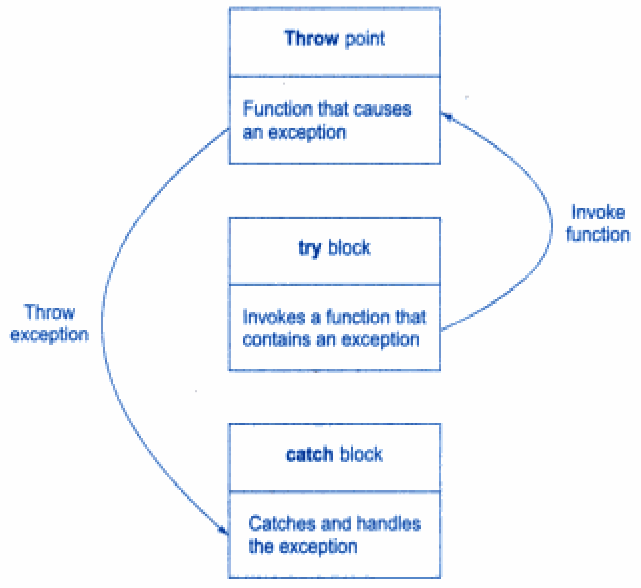
Syntax:

```
try
{
    throw exception;
}
catch(type arg)
{
    //some code
}
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    try {
        throw 6;
    }

    catch (int a) {
        cout << "An exception occurred!" <<
endl;
        cout << "Exception number is: " << a <<
endl;
    }
  }
```

The exceptions are thrown by functions that are invoked from within the try blocks. The point at which the throw is executed is called the throw point. Once an exception is thrown to the catch block, the control cannot return to the throw point, This relationship is shown in figure.


Throw point
Function that causes an exception
Invoke function
try block
Invokes a function that contains an exception
Throw exception
catch block
Catches and handles the exception

Function invoked by try block throwing exception

Multiple catch statements
A single try statement can have multiple catch statements. Execution of particular catch block depends on the type of exception thrown by the throw keyword. If throw keyword send exception of integer type, catch block with integer parameter will get execute.
Example of multiple catch blocks

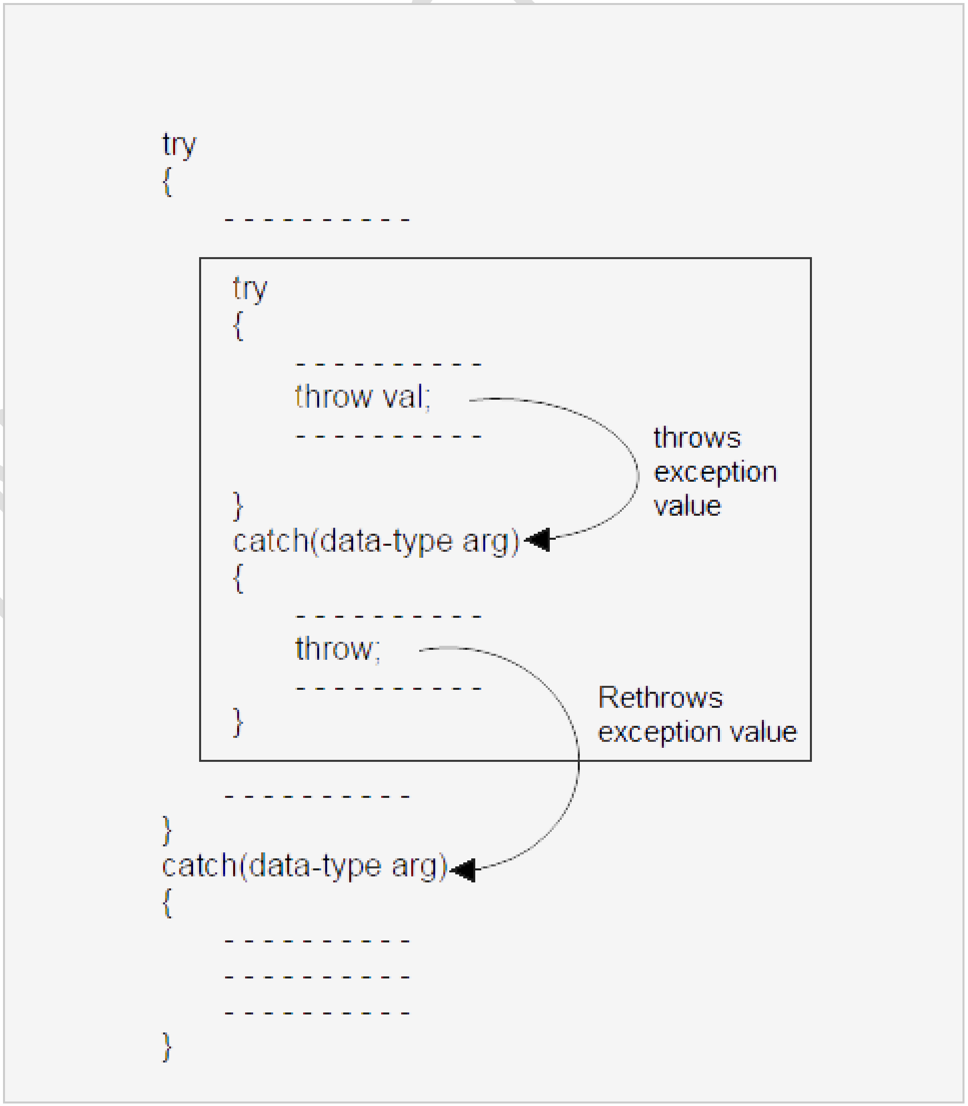
```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a=2;

      try
      {
         if(a==1)
```

```
                throw a;            //throwing integer exception

            else if(a==2)
                throw 'A';          //throwing character
exception

            else if(a==3)
                throw 4.5;          //throwing float exception

        }
        catch(int a)
        {
            cout<<"\nInteger exception caught.";
        }
        catch(char ch)
        {
            cout<<"\nCharacter exception caught.";
        }
        catch(double d)
        {
            cout<<"\nDouble exception caught.";
        }

        cout<<"\nEnd of program.";

    }

  Output :

        Character exception caught.
        End of program.
```

Rethrowing Exceptions

Rethrowing exception is possible, where we have an inner and outer try-catch statements (Nested try-catch). An exception to be thrown from inner catch block to outer catch block is called rethrowing exception.

Syntax of rethrowing exceptions



```
try
{
    ----------
    try
    {
        ----------
        throw val;
        ----------
    }
    catch(data-type arg)
    {
        ----------
        throw;
        ----------
    }
    ----------
}
catch(data-type arg)
{
    ----------
    ----------
    ----------
}
```

Example of rethrowing exceptions

```
    #include<iostream.h>
```

```
#include<conio.h>
void main()
{
    int a=1;
      try
      {
            try
            {
                  throw a;
            }
            catch(int x)
            {
               cout<<"\nException in inner try-catch block.";
               throw x;
            }

      }
      catch(int n)
      {
         cout<<"\nException in outer try-catch block.";

      }

      cout<<"\nEnd of program.";

}
```

Output :

```
Exception in inner try-catch block.

Exception in outer try-catch block.

End of program.
```

UNIT 4 COMPLETED

COMPONENTS OF STL

CONTAINERS

Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.

Following are the containers that give the details of all the containers as well as the header file and the type of iterator associated with them :

Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional

multiset	Multiset is an associate container for storing non- unique sets.	<set>	Bidirectional
map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key- value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator

Classification of containers :

- Sequence containers
- Associative containers
- Derived containers

Note : Each container class contains a set of functions that can be used to manipulate the contents.

ITERATOR

- Iterators are pointer-like entities used to access the individual elements in a container.
- Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.
- Iterator contains mainly two functions:

begin(): The member function begin() returns an iterator to the first element of the vector.

end(): The member function end() returns an iterator to the past-the-last element of a container.

Iterator Categories

Iterators are mainly divided into five categories:

Input iterator:

- An Input iterator is an iterator that allows the program to read the values from the container.
- Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.
- An Input iterator is a one way iterator.
- An Input iterator can be incremented, but it cannot be decremented.

2. **Output iterator:**
    - An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.
    - It is a one-way iterator.
    - It is a write only iterator.

3. **Forward iterator:**
    - Forward iterator uses the ++ operator to navigate through the container.
    - Forward iterator goes through each element of a container and one element at a time.
4. **Bidirectional iterator:**
    - A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.
    - It is a two way iterator.
    - It can be incremented as well as decremented.
5. **Random Access Iterator:**
    - Random access iterator can be used to access the random element of a container.
    - Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.

Operations supported by iterators

iterator	Element access	Read	Write	Increment operation	Comparison
Input	->	v = *p		++	==,!=
output			*p = v	++	
forward	->	v = *p	*p = v	++	==,!=
Bidirectional	->	v = *p	*p = v	++,--	==,!=

Random access	->,[ ]	v = *p	*p = v	++,--,+,-,+=,--=	==,!=,<,>,<=,>=

Algorithms

Algorithms are the functions used across a variety of containers for processing its contents.

Points to Remember:

- Algorithms provide approx 60 algorithm functions to perform the complex operations.
- Standard algorithms allow us to work with two different types of the container at the same time.
- Algorithms are not the member functions of a container, but they are the standalone template functions.
- Algorithms save a lot of time and effort.
- If we want to access the STL algorithms, we must include the <algorithm> header file in our program.

STL algorithms can be categorized as:

Nonmutating algorithms: Nonmutating algorithms are the algorithms that do not alter any value of a container object nor do they change the order of the elements in which they appear. These algorithms can be used for all the container objects, and they make use of the forward iterators.

- **Mutating algorithms**: Mutating algorithms are the algorithms that can be used to alter the value of a container. They can also be used to change the order of the elements in which they appear.
- **Sorting algorithms**: Sorting algorithms are the modifying algorithms used to sort the elements in a container.
- **Set algorithms**: Set algorithms are also known as sorted range algorithm. This algorithm is used to perform some function on a container that greatly improves the efficiency of a program.
- **Relational algorithms**: Relational algorithms are the algorithms used to work on the numerical data. They are mainly designed to perform the mathematical operations to all the elements in a container.

FUNCTION OBJECTS

A Function object is a function wrapped in a class so that it looks like an object. A function object extends the characteristics of a regular function by using the feature of aN object oriented such as generic programming. Therefore, we can say that the function object is a smart pointer that has many advantages over the normal function.

Following are the advantages of function objects over a regular function:

- Function objects can have member functions as well as member attributes.
- Function objects can be initialized before their usage.
- Regular functions can have different types only when the signature differs. Function objects can have different types even when the signature is the same.
- Function objects are faster than the regular function.

A function object is also known as a '**functor**'. A function object is an object that contains atleast one definition of **operator()**function. It means that if we declare the object 'd' of a class in which **operator()** function is defined, we can use the object 'd' as a regular function.

Suppose 'd' is an object of a class, operator() function can be called as:

d();

which is same as:

d.operator() ( );

simple example:

```
 #include <iostream>
 using namespace std;
class function_object
{
      public:
     int operator()(int a, int b)
    {
        return a+b;
    }
};

 int main()
{
   function_object f;
   int result = f(5,5);
  cout<<"Addition of a and b is : "<<result;

return 0;
}
```

Output:

```
Addition of a and b is : 10
```

In the above example, 'f' is an object of a function_object class which contains the definition of operator() function. Therefore, 'f' can be used as an ordinary function to call the operator() function.

Manipulating strings

String Class in C++

- Creating string objects
- Reading string objects from keyboard

- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string
- Adding objects of string
- Comparing strings
- Accessing characters of a string
- Obtaining the size or length of a string, etc...

Manipulating Null terminated Strings

- strcpy(str1, str2): Copies string str2 into string str1.
- strcat(str1, str2): Concatenates string str2 onto the end of string str1.
- strlen(str1): Returns the length of string str1.
- strcmp(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- strchr(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- strstr(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- append(): This function appends a part of a string to another string
- assign():This function assigns a partial string
- at(): This function obtains the character stored at a specified location
- begin(): This function returns a reference to the start of the string
- capacity(): This function gives the total element that can be stored
- compare(): This function compares a string against the invoking string

- empty(): This function returns true if the string is empty
- end(): This function returns a reference to the end of the string
- erase(): This function removes character as specified
- find(): This function searches for the occurrence of a specified substring
- length(): It gives the size of a string or the number of elements of a string
- swap(): This function swaps the given string with the invoking one

Operators used for String Class

1. =: assignment
2. +: concatenation
3. ==: Equality
4. !=: Inequality
5. <: Less than
6. <=: Less than or equal
7. >: Greater than
8. >=: Greater than or equal
9. []: Subscription
10. <<: Output
11. >>: Input