8.11 File Access Pointers ... 17
8.12 Sequential and Random Access to File ... 17
8.13 Testing Errors during File Operations ... 17
Some Examples ... 17

Chapter 9

Templates

9.1 Function Templates ... 20
9.2 Overloading Function Template ... 20
9.3 Class Template ... 20
9.4 Derived Class Template ... 21
9.5 Introduction to Standard Template Library ... 21
Some Examples ... 22

Chapter 10:

Exception Handling

10.1 Error Handling ... 234
10.2 Exception Handling Constructs (Try, Catch, Throw) ... 234
10.3 Advantage over Conventional Error Handling ... 236
10.4 Multiple Exception Handling ... 237
10.5 Re-throwing Exception ... 238
10.6 Catching all Exception ... 239
10.7 Exception with Arguments ... 239
10.8 Exception Specification for Function ... 241
10.9 Handling Uncaught and Unexpected Exceptions ... 242
Some Examples ... 244

CHAPTER 1:

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

1.1 Issues with Procedure Oriented Programming

1. Emphasis done on algorithm (or procedure) rather than data.
2. Large programs are divided into smaller programs known as functions. Change in a data type being processed needs to propagate to all the functions that use the same data type.
3. In large program, it is difficult to identify belonging of global data.
4. Maintaining and enhancing program code is difficult because of global data.
5. It does not model real world very well.

1.2 Basics of Object Oriented Programming (OOP)

Why is object oriented programming necessary in programming?

To remove some of the flaws encountered in procedural approach, object-oriented approach has been invented.

OOP treats data as a critical element in the program development and does not make to flow freely around the program. It tries data more closely to functions that operate on it and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.

The focus of procedural programming is to break down a programming task into a collection of data structures and routines where in OOP, it is to breakdown into objects.

What type of language is C++?

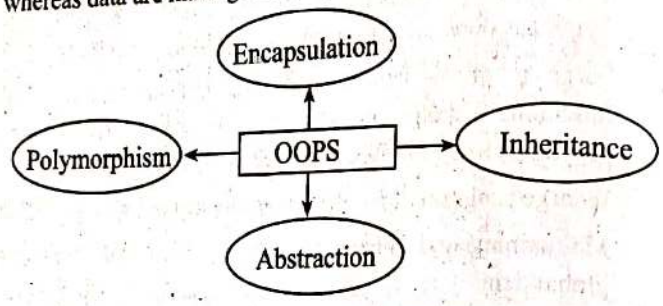
C++ is object oriented programming language.

Features of object oriented programming

i. Emphasis is on data rather than procedure.
ii. Programs are divided into objects.
iii. An object is a complete entity i.e., it has all the data and associated functions within it. Whenever, something is to be changed for an object, only its class gets affected because it is complete in itself.

All the functions that are working on this data or using it are defined within the class; they get to see the change immediately. And nowhere else change is required.

iv. Data is hidden and cannot be accessed by external functions whereas data are made global in procedure oriented programming.



1.3 Procedure Oriented versus Object Oriented Programming

Procedural Programming	Object Oriented Programming
• It emphasizes on doing the things only i.e., it focuses on algorithm.	• It emphasizes on data rather than procedure or algoritm.
• Larger programs are divided into smaller programs known as functions.	• Programs are divided into what we know as object.
• Most of the function share the global data. Function transforms the data from one to another.	• Object may communicate with each other through function.
• It doesnot model real world problems.	• It models the real world problems.
• Code reusability is difficult.	• Code reusability is easy as compare to Procedural Programming.
• E.g., C follows top down approach.	• E.g., C++ follows bottom-up approach.

1.4 Concept of Object Oriented Programming

1.4.1 Object

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data, or any item that the program has to handle. They may also represent user-defined data such as vectors, time, and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Object has its own characteristics and behavior. The characteristics of an object is represented by data while its behavior is represented by function. In fact, objects are variables of the type class.

1.4.2 Class

A class is a collection of objects of similar type. For example, mango, apple, and orange are members of the class fruit. Once a class is created, we can create any number of objects belonging to that class.

1.4.3 Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes.

1.4.4 Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.

Its advantages are that the data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

1.4.5 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. In OOP, the concept of inheritance provides the idea of reusability. This means we can add additional features to an existing class without modifying it.

1.4.6 Polymorphism

Polymorphism means the ability to take more than one form. The function overloading and operator overloading show the concept of compile time polymorphism while run time polymorphism is achieved by using virtual function.

1.5 Example of Some Object Oriented Languages

An object-oriented programming language is one that follows object-oriented programming techniques such as encapsulation, inheritance, polymorphism. In around 1972-1980, a pure object-oriented programming language **Smalltalk** was developed. It was called pure object oriented language because everything in them was treated as objects. It was designed specifically to facilitate and enforce Object Oriented methods. Some language like **Java, Python** were designed mainly for Object Oriented programming along with some procedural elements. Apart from this, some languages like **C++, Perl** are historically procedural languages, but have been extended with some object oriented features. Besides these, there are some languages that support abstract data type but not all features of object oriented programming. They are called object-based languages. Example of such language are **Modula-2, Pliant, Ada.**

1.6 Advantages and Disadvantages of OOP

Advantages

1. Emphasis is on data rather than procedure.
2. Programs are divided into objects.
3. An object is a complete entity i.e. it has all the data and associated functions within it. Whenever, something is to be changed for an object, only its class gets affected because it is complete in itself. All the functions that are working on this data or using it are defined within the class; they get to see the change immediately. And nowhere else change is required.
4. Data is hidden and cannot be accessed by external functions whereas data are made global in procedure oriented programming.

Disadvantages

1. Compiler and runtime overhead is high because object oriented program requires more time during compilation and for dynamic and runtime support it requires more resource and processing time.
2. Re-orientation of software developer to object oriented thinking.
3. Requires the mastery in software engineering and programming methodology.
4. Benefits only in long run while managing large software projects.
5. The message passing between many objects in a complex application can be difficult to trace & debug.

SOME EXAMPLES

1. Example of object oriented programming. [2067 Ashad]

```
#include<iostream>
using namespace std;
class time
{
  int hr, min, sec; //declaration of hour, minutes and second.
  public:
   void input( )
   {
        cout<<"Enter hour, minutes and seconds";
        cin>>hr>>min>>sec;
   }
   void display( )
   {
        cout<<hr<<"Hour,"<<min<<"Minutes and "<<sec<<
        "Second" <<endl;
   }
   void add(time t1,time t2)
   {
        sec=t1.sec+t2.sec;
        min=t1.min+t2.min+sec/60;
        sec=sec%60;
        hr=t1.hr+t2.hr+min/60;
        min=min%60;
   }
   void subtract(time t1, time t2)
   {
        min=0;
        hr=0;
        sec=t2.sec-t1.sec;
```

```
        if(sec<0)
        {
        sec=sec+60;
        min=min-1;
        }
        min=min+t2.min-t1.min;
        if(min<0)
        {
        min=min+60;
        hr=hr-1;
        }
        hr=hr+t2.hr-t1.hr;
    }
};
int main( )
{
    time t1,t2,t3,t4;
    t1.input( );
    t2.input( );
    t3.add(t1,t2);
    cout<<endl<<"The addition of two time is:";
    t3.display( );
    t4.subtract(t1,t2);
    cout<<endl<<"The subtraction of two time is:";
    t4.display( );
    return 0;
}
```

2. Write a program to find prime number in procedural and object oriented ways. [2070 Ashad]

Procedural Method:

```
#include<iostream>
using namespace std;
int main( )
{
 int no,count=0,i;
 printf("Enter the number:\n)";
 scanf("%d",no);
 for(i=1;i<=no;i++)
 {
     if(no%i==0)
     count++;
 }
 if(count==2)
 printf("The number is prime.\n");
 else
 printf("The number is not prime.\n");
   return 0;
}
```

Object Oriented Method:

```
#include<iostream>
using namespace std;
int main( )
{
 int no,count=0;
 cout<<"Enter the number:"<<endl;
 cin>>no;
 for(int i=1;i<=no;i++)
 {
       if(no%i==0)
       count++;
 }
 if(count==2)
 cout<<"The number is prime."<<endl;
 else
 cout<<"The number is not prime."<<endl;
   return 0;
   }
```

```
        if(sec<0)
        {
        sec=sec+60;
        min=min-1;
        }
        min=min+t2.min-t1.min;
        if(min<0)
        {
        min=min+60;
        hr=hr-1;
        }
        hr=hr+t2.hr-t1.hr;
    }
};
int main( )
{
    time t1,t2,t3,t4;
    t1.input( );
    t2.input( );
    t3.add(t1,t2);
    cout<<endl<<"The addition of two time is:";
    t3.display( );
    t4.subtract(t1,t2);
    cout<<endl<<"The subtraction of two time is:";
    t4.display( );
    return 0;
}
```

2. Write a program to find prime number in procedural and object oriented ways. [2070 Ashad]

Procedural Method:

```
#include<iostream>
using namespace std;
int main( )
{
 int no,count=0,i;
 printf("Enter the number:\n)";
 scanf("%d",no);
 for(i=1;i<=no;i++)
 {
     if(no%i==0)
     count++;
 }
 if(count==2)
 printf("The number is prime.\n");
 else
 printf("The number is not prime.\n");
   return 0;
}
```

Object Oriented Method:

```
#include<iostream>
using namespace std;
int main( )
{
  int no,count=0;
  cout<<"Enter the number:"<<endl;
  cin>>no;
  for(int i=1;i<=no;i++)
  {
        if(no%i==0)
        count++;
  }
  if(count==2)
  cout<<"The number is prime."<<endl;
  else
  cout<<"The number is not prime."<<endl;
     return 0;
     }
```

CHAPTER 2:

INTRODUCTION TO C++

2.1 The Need of C++

1. C++ is a highly portable language and is often the language of choice for multi-device, multi-platform app development.
2. C++ is an object-oriented programming language and includes classes, inheritance, polymorphism, data abstraction and encapsulation.
3. C++ has a rich function library.
4. C++ allows exception handling, and function overloading which are not possible in C.
5. C++ is a powerful, efficient and fast language. It finds a wide range of applications – from GUI applications to 3D graphics for games to real-time mathematical simulations.

2.2 Features of C++

C++ consists of common features of object oriented language as well as some typical features possessed by C. Following are some important features of C++:

Inline function:

To save the execution time for small functions, we can put the code of the function body directly in the line of called location. Meaning that, in each function call the codes in the called function get inserted at the line of the call instead of jumping from calling function to called function and back to calling function again. This type of function whose code is copied in the code location is called inline function. This feature reduces switching overhead and makes the execution of the program faster at the cost of program size.

Operator overloading:

The operator overloading feature of C++ extends the meaning of an operator for user defined types. The operators such as +, -, +=, >, <<, etc are designed to operate only on standard data types. However, through the operator overloading feature we can extend the meaning of the operators to operate on user defined types. For example the operator + can be used to perform the addition operation on user defined data such as string concatenation, complex number addition etc.

Run time polymorphism:

In C++, the member function belonging to class can be selected at runtime. C++ has features like classes, virtual functions, operator overloading, inheritance, templates, exception handling, STL, RTTI and many more. C++ provides stronger type checking than C and directly supports a wider range of programming style than C.

2.3 C++ versus C

C	C++
1. C follows the procedural programming paradigm i.e., importance is given to the steps or procedure of the program.	1. C++ is a multi-paradigm language (procedural as well as object oriented) i.e., C++ focuses on the data rather than the procedures.
2. C uses the top-down approach.	2. C++ uses the bottom-up approach.
3. C applications are faster to compile than C++ application.	3. C++ application are comparatively slower to compile than C.
4. Functions are the building blocks of a C program and so it is function-driven.	4. C++ is object-driven, as objects are building blocks of a C++ program.

2.4 History of C++

The C++ programming language has a history going back to 1979, when Bjarne Stroustrup was doing work for his Ph.D. thesis. One of the languages Stroustrup had the opportunity to work with was a language called Simula, which as the name implies is a language primarily designed for simulations. The Simula 67 language - which was the variant that Stroustrup worked with - is regarded as the first language to support the object-oriented programming paradigm. Stroustrup found that this paradigm was very useful for software development, however the Simula language was far too slow for practical use.

Shortly thereafter, he began work on "C with Classes", which as the name implies was meant to be a superset of the C language. His goal was to add object-oriented programming into the C language, which was

and still is a language well-respected for its portability without sacrificing speed' or low-level functionality. His language included classes, basic inheritance, inlining, default function arguments, and strong type checking in addition to all the features of the C language.

The first C with Classes compiler was called Cfront, which was derived from a C compiler called CPre. It was a program designed to translate C with Classes code to ordinary C. A rather interesting point worth noting is that Cfront was written mostly in C with Classes, making it a self-hosting compiler (a compiler that can compile itself). Cfront would later be abandoned in 1993 after it became difficult to integrate new features into it, namely C++ exceptions. Nonetheless, Cfront made a huge impact on the implementations of future compilers and on the Unix operating system.

In 1983, the name of the language was changed from C with Classes to C++. The ++ operator in the C language is an operator for incrementing a variable, which gives some insight into how Stroustrup regarded the language. Many new features were added around this time, the most notable of which are virtual functions, function overloading, references with the & symbol, the const keyword, and single-line comments using two forward slashes (which is a feature taken from the language BCPL).

In 1985, Stroustrup's reference to the language entitled *The C++ Programming Language* was published. That same year, C++ was implemented as a commercial product. The language was not officially standardized yet, making the book a very important reference. The language was updated again in 1989 to include protected and static members, as well as inheritance from several classes.

In 1990, *The Annotated C++ Reference Manual* was released. The same year, Borland's Turbo C++ compiler would be released as a commercial product. Turbo C++ added a plethora of additional libraries which would have a considerable impact on C++'s development. Although Turbo C++'s last stable release was in 2006, the compiler is still widely used.

In 1998, the C++ standards committee published the first international standard for C++ ISO/IEC 14882:1998, which would be informally known as C++98. *The Annotated C++ Reference Manual* was said to be a large influence in the development of the standard. The Standard Template Library, which began its conceptual development in 1979, was also included. In 2003, the committee responded to multiple problems that were reported with their 1998 standard, and revised it accordingly. The changed language was dubbed C++03.

In 2005, the C++ standards committee released a technical report (dubbed TR1) detailing various features they were planning to add to the latest C++ standard. The new standard was informally dubbed C++0x as it was expected to be released sometime before the end of the first decade. Ironically, however, the new standard would not be released until mid-2011. Several technical reports were released up until then, and some compilers began adding experimental support for the new features.

In mid-2011, the new C++ standard (dubbed C++11) was finished. The Boost library project made a considerable impact on the new standard, and some of the new modules were derived directly from the corresponding Boost libraries. Some of the new features included regular expression support (details on regular expressions may be found here), a comprehensive randomization library, a new C++ time library, atomics support, a standard threading library (which up until 2011 both C and C++ were lacking), a new for loop syntax providing functionality similar to foreach loops in certain other languages, the auto keyword, new container classes, better support for unions and array-initialization lists, and variadic templates.

CHAPTER 3:

C++ LANGUAGE CONSTRUCTS

3.1 C++ Program Structure

Include files
Class declaration
Member functions definitions
Main function programs

Fig.: C++ program structure

3.2 Character Sets and Token

Compile collects the valid character sets to make sensible tokens. So, it can be said that tokens are collection of different characters, symbols, operators or punctuators.

Following are different types of tokens identified by C++ systems:

1. **Keywords**

   They are reserved words and cannot be used for any other purpose.

2. **Identifiers**

   They are programmer defined elements like the name of the variables, functions, arrays, structure etc. These are tokens which are sequence of letters, digits and underscore (_).

3. **Constants**

   Constants, also called literals, are the actual representation of values used in program. Types of constants are: integer constants, floating point constants, character constants, string constants, and enumerated constants.

4. **Operators**

   C++ language has different character sets and symbols. Operators are special symbols which cause to perform mathematical or logical operations.

5. **Punctuators**

   The special symbols which acts like punctuation marks are called punctuators.

3.3 Variable Declaration and Expression

Variable Declaration

Variables are entities which holds values and whose value may change throughout program execution. Each and every variables must be declared before hand it is used.

```
data type variable_name1, variable_name2,...,variable_nameN;
E.g.,     int length, breadth;
          float area;
          char section;
```

Also, variables can be initialized during declaration as:

```
int length =5, breadth = 3;
float area = 14.57;
char section = 'A';
```

Expression:

Any arrangement of variables, constants and operators that specifies a computation is called an expression. Thus,

alpha + 12 and **(alpha-37)*beta/2** are expressions.

Parts of expressions may also be expression. In second example, alpha-37 and beta/2 are both expressions.

Note: The expressions aren't the same as statements. The statements terminate with a semicolon. There can be several expressions in a statement.

Types of expressions:

Type	example
Constant expressions	20+4/2, 'X'
Integral expressions	m*n-3 ; m and n are integers
Float expressions	x+y/2.0; x and y are floats
Pointer expressions	&m, ptr +1 ; m is a variable, ptr is pointer
Relational expressions	X<=Y, etc
Logical expressions	A>B && X==10
Bitwise expressions	X<<3

3.4 Statements

Statements are the instructions given to the computer to perform any kind of action, it can be evaluating expression or making decision or repeating action. A statement forms a smallest executable unit within a

C++ program. Statements are terminated with a semicolon (;). The simplest statement is the empty or null statement. General syntax is:

Sum = a + b;

3.5 Data Type

type	byte	range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
long int	4	
float	4	
double	8	
long double	10	

C++ Data Types

i. **User defined type**
- structure
- union
- class
- enumeration

ii. **Derived type**
- array
- pointer
- reference

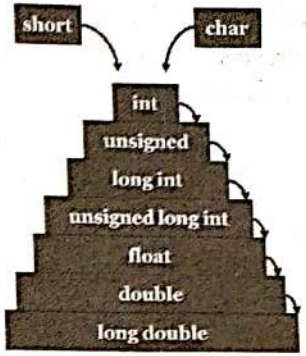
iii. **Built-in type**
- void
- integral type
  - int
  - char
- floating type
  - float
  - double

3.6 Type Conversion and Promotion Rules

Implicit Type Conversion

Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is called implicit or automatic type conversion.

For a binary operator, if the operands type differ the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type.



For example, if one of the operand is an 'int' and the other is a 'float', the int is converted into a float because a 'float' is wider than an 'int'.

Explicit Type Conversion

C++ permits explicit type conversion of variables using the type cast operator. Hence, also called 'cast'. This conversion is done by programmers as per need.

(type name) expression //c notation

type name (expression) //c++ notation

E.g.,

average = sum/(float) i ; //c

average = sum / float (i); //c++

ANSI C++ adds the following new cast operators

const_cast

static_cast

dynamic_cast

reinterpret_cast

3.7 Preprocessor Directives

The lines beginning with hash (#) sign are called preprocessor directive.

A preprocessor directive is an instruction to the compiler (they are not program statements).

A part of the compiler called the preprocessor deals with these directives before it begins the real compilation process.

E.g.,

#include<iostream>

#define PI 3.1415 etc

Note: They do not terminate with a semicolon

3.8 Namespace

The namespace mechanism is used for the logical grouping of program elements like variables, classes, functions etc. The purpose of namespace is to localize a name of identifiers to get rid of naming conflicts across different modules designed by different members of programming team.

Syntax for defining the namespace is as follows:

```
namespace namespace_name
{
        //Declaration of variable, function, class, etc.
}       //no semicolon at the end
using namespace namespace_name;
usingnamespace_name::member;
```

The first form includes all the members of the namespace scope to our scope where as second includes only the specified members into our scope.

Program:

```
#include<iostream>
using namespace std;
namespace convert
{
void volume(int& a)
  {
a=a*100;
  }
}
int main( )
{
int no;
cout<<"Enter meter:"<<endl;
cin>>no;
convert::volume(no);
cout<<"The convertion is:"<<no;
return 0;
}
```


Significance of namespace:

Namespace have following important points:

1. We can have more than one namespace of the same name. This gives the advantage of defining the same namespace in more than one file (although they can be created in the same file as well).

   For example,

```
#include<iostream>
using namespace std;
namespace mynamespace
{
    int x;
}
namespace mynamespace
{
    int y;
}
int main(int argc, char * argv[ ])
{
mynamespace::x = mynamespace::y =5;
cout << mynamespace::x << endl << mynamespace::y;
}
```

2. We can have anonymous namespaces (namespace with no name). They are directly usable in the same program and are used for declaring unique identifiers. It also avoids making global static variable.
3. C++ has a default namespace named std, which contains all the default library of the C++ included using #include directive.

3.9 User Defined Constant const

The statement

```
Float PI=3.1452;
```

defines a variable named PI, and is assigned with the floating-point numeric constant value 3.1452. In the above case, the variable PI is considered as a constant, whose value doesnot change throughout the life of the program. However, an accidental change of the value of the variable PI is not restricted by C.

C++ overcomes this by supporting a new constant qualifier for definition. The qualifier for defining a variable, whose value cannot be changed once it is assigned with a value cannot be changed once it is assigned with a value at the time of variable definition. The qualifier used in C++ to define such variables is the const qualifier. Any attempt to change the value of variable defined with qualifier will give an error.

Syntax: const [DataType] Variable Name = constant value

E.g.,

```
const int TRUE =1;
const char *book_name="OOPS with C++";
```

Program Example

```
#include<iostream>
using namespace std;
const float PI=3.1452;
int main( )
{ float radius, area;
cout<<"Enter Radius of Circle:"; cin>>radius;
area = PI*radius*radius;
cout<<"Area of Circle ="<<area;
return 0;
}
```

In the program, if we modify a constant type variable, then it leads to the compilation error: cannot modify a *const* object.

3.10 Input/Output Streams and Manipulators

Input Stream

Input stream is a sequence of characters from input device to the computer. The input stream allows us to perform read operations with input devices such as keyboard, disk etc. Input is performed using the *cin* object. The cin is predefined object of the library class istream to refer the input device. The syntax for the standard input stream operation is as follows:

```
cout<<varname;
```

Output Stream

Output stream is a sequence of characters from the computer to output device. The output stream allows us to write operations on standard output devices. The output operation is performed by making use of the obect *cout*. The cout is predefined object of the library class ostream to refer the output device. The syntax for the standard output stream operation is as follows:

```
cin>>varname;
```

3.11 Dynamic Memory Allocation with New and Delete

New Operator

C++ provides new approach to obtaining blocks of memory using the new operator. This operator obtains memory from the operating system and returns a pointer to its starting point.

Example:

```
#include<iostream>
using namespace std;
int main( )
{
char *str="Idle hands are the devil's workshop.";
int len = strlen(str);          //get length of str
char *ptr;                      //make a pointer to char
ptr = new char[len+1];          //set aside memory:string + '\0'
strcpy(ptr,str);                //copy str to new memory area ptr
cout<<endl<<"ptr="<<ptr;        //show that str is now in ptr
delete ptr;
return 0;
}
```

The expression

```
ptr=new char[len + 1];
```

returns a pointer that points to a section of memory just large enough to hold the string str, whose length len we found with the strlen( ) function, plus an extra byte for the null character '\0' at the end of the string.

The Delete Operator

If our program reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system.

The expression

```
delete ptr;
```

releases the memory block pointed by ptr.

3.12 Condition and Looping

Looping in C++ are:

The While Loop

The simplest kind of loop is the while-loop. Its syntax is:

```
while (expression) statement
```

The while-loop simply repeats statement while expression is true. If, after any execution of statement, expression is no longer true, the loop ends, and the program continues right after the loop. For example,

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ( )
{
 int n = 10;
 while (n>0) {
  cout << n << ", ";
  --n;
 }
cout << "liftoff!\n";
return 0;
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!

The Do-While Loop

A very similar loop is the do-while loop, whose syntax is:

```
do statement while (condition);
```

It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled.

For example,

```
// echo machine
#include <iostream>
#include <string>
using namespace std;
int main( )
{
 string str;
 do {
  cout << "Enter text: ";
  getline (cin,str);
  cout << "You entered: " << str << '\n';
 }while (str != "goodbye");
 return 0;
}
```

Output:

```
Enter text: hello
You entered: hello
Enter text: who's there?
You entered: who's there?
Enter text: goodbye
You entered: goodbye
```

The For Loop

The for loop is designed to iterate a number of times. Its syntax is:

```
for (initialization; condition; increase) statement;
```

Like the while loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization, condition and an increase expression, respectively. Therefore, it is especially useful to use counter variables as condition.

It works in the following way:

1. Initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. Condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
3. Statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. Increase is executed, and the loop gets back to step 2.
5. The loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ( )
{
 for (int n=10; n>0; n--)
 {
  cout << n << ", ";
 }
 cout << "liftoff!\n";
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!

3.13 Functions

3.13.1 Function Syntax

General syntax:

```
return_type fxn_name (arg1,arg2,.....)      //function body
 {

                                            // body

 }
```

```
int main( )
 {
  fxn_name(args);                           // calling function
 }
```

3.13.2 Function Overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters. This capability is called function overloading. An overloaded functions appears to perform different activities depending on the kind of data sent to it. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that performs similar tasks, but on different data types.

Ambiguity and its Resolution

Ambiguity in function overloading:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as

   char to int

   float to double
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   long square(long n);

   double square (double x);

   A function call such as

   square (10);

   will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.

It works in the following way:

1. Initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. Condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
3. Statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. Increase is executed, and the loop gets back to step 2.
5. The loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ( )
{
 for (int n=10; n>0; n--)
 {
  cout << n << ", ";
 }
 cout << "liftoff!\n";
}
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!

3.13 Functions

3.13.1 Function Syntax

General syntax:

```
return_type fxn_name (arg1,arg2,.....)      //function body
 {

                                            // body
 }
int main( )
 {
  fxn_name(args);                           // calling function
 }
```

3.13.2 Function Overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters. This capability is called function overloading. An overloaded functions appears to perform different activities depending on the kind of data sent to it. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that performs similar tasks, but on different data types.

Ambiguity and its Resolution

Ambiguity in function overloading:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as

   char to int

   float to double
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   long square(long n);

   double square (double x);

   A function call such as

   square (10);

   will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.

Function overloading with default argument:

The overloaded function have default argument in the following ways:

In C, when a function is called, the number of actual and formal parameters must be same. In C++, there is a provision of supplying less number of arguments than the actual number of parameters. This mechanism is supported by the default argument. If we do not supply any argument the default value is used to assign the value that is absent in the function call.

a. The default value are specified when function is declared.
b. The default value is specified similar to the variable initialization.
c. The default values can be constants, global variables, or even a function call.
d. All of the default argument must be in the right part of parameter list. Suppose a function has more than one default arguments.
e. In a function call, if a value of a default argument is not specified, than the arguments from right are omitted.

Some example of function declared with default value:

```
float interest (float p, int time, float rate=0.14);          //ok
float interest (float p, int time=10, float rate);            //error
float interest (float p, int time=5, float rate=0.15);        //ok
float interest (float p=1000, int time, float rate=0.12);//error
```

Program Code:

```
#include<iostream>
using namespace std;
int simple_interest(int p=100, int t=2, int r=10)
{
  return((p*t*r)/100);
}
int main( )
{
int principle, time, rate;
cout<<"Default simple interest is:"<<simple_interest()<<endl;
cout<<"Enter principle:"<<endl;
cin>>principle;
cout<<"One argument simple interest is:"<<simple_interest (principle)<<endl;
cout<<"Enter principle and time:"<<endl;
cin>>principle>>time;
cout<<"Two argument simple interest is:"<<simple_interest (principle, time)<<endl;
cout<<"Enter principle, time, and rate:"<<endl;
cin>>principle>>time>>rate;
cout<<"Three argument simple interest is:"<<simple_interest (principle,time,rate)<<endl;
return 0;
}
```

3.13.3 Inline Functions

An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code. To make a function inline, prefix the keyword *inline* to the function definition.

Advantages:

Whenever function is called, it takes lots of extra time to executing a series of instructions for tasks such as jumping to function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads. To eliminate the cost of calls to small functions, we use the concept of inline functions.

Disadvantages:

i. For functions returning values, if a loop, a switch, or a goto exists.
ii. For functions not returning values, if a return statements exists.
iii. If functions contain static variables.
iv. If inline functions are recursive.

Differentiation of inline function with macro:

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is

likely to be called many times. However, every times a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function.

An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code. To make a function inline, prefix the keyword *inline* to the function definition.

Whenever function is called, it takes lots of extra time to executing a series of instructions for tasks such as jumping to function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads. To eliminate the cost of calls to small functions we use the concept of inline functions.

C++ supports one more type of variable called reference variable, in addition to the value variable and pointer variables of C. Value variables are used to hold some numeric values; pointer variables are used to hold the address of some other value variables. Reference variable behaves similar to both, a value variable and a pointer variable. In program code, it is used similar to that of a value variable, but has an action of a pointer variable. Thus, a reference variable provides an alias (alternative name) of the variable that is previously defined. In C, the corresponding parameter in the calling function must be declared as a pointer type. In C++, the corresponding parameter can be declared as any reference type, not just a pointer type.

In pointer variable, the returning from the function will be the value while in reference variable, in case of returning the value, it will return the variable.

For example,

```
#include <iostream>
using namespace std;
int main( )
{
int x=5;
int &y=x;
cout<<"X="<<x<<"and Y="<<y<<endl;
y++;
cout<<"X="<<x<<"and Y="<<y<<endl;
return 0;
}
```

Ouput: X=5 and Y=5
X=6 and Y=6

Program code:

```
#include<iostream>
using namepspace std;
int simple_interest(int p=100, int t=2, int r=10)
{
return((p*t*r)/100);
}
int main( )
{
int principle, time, rate;
cout<<"Default simple interest is:"
<<simple_interest( )<<endl;
cout<<"Enter principle:"<<endl;
cin>>principle;
cout<<"One argument simple interest is:"
<<simple_interest (principle)<<endl;
cout<<"Enter principle and time:"<<endl;
cin>>principle>>time;
cout<<"Two argument simple interest is:"
<<simple_interest (principle,time)<<endl;
cout<<"Enter principle, time, and rate:"<<endl;
cin>>principle>>time>>rate;
cout<<"Three argument simple interest is:"
<<simple_interest (principle,time,rate)<<endl;
return 0;
}
```

3.13.4 Default Argument

C++ allow us to assign default value(s) to a function's parameter(s), which is useful in case a matching argument is not passed in the function call statement. The default values are specified at the time of function declaration.

Example: float interest(float p, int time, float r=0.1);

The above function declaration provides default value of 0.1 to the argument r. In above function if the function call is interest (500,5) then r value will be taken as 0.1 and if the function call is interest(500,5,0.15) then the value of r will be 0.15.

Program code:

```
#include<iostream.h>
#include<math.h>
using namepspace std;
void area(float a,floatb,float c=7)
{
float s=(a+b+c)/2;
float ar=sqrt(s*(s-a)*(s-b)*(s-c));
cout<<"\nArea of triangle is:"<<ar;
}
void area(int b,int h)
{
float ar=(b*h)/2;
cout<<"\nArea of triangle is:"<<ar;
}
int main( )
{
float side1,side2,side3;
int b,h;
cout<<"\nEnter three side of triangle:";
cin>>side1>>side2>>side3;
area(side1,side2,side3);
cout<<"\nEnter base and height of triangle:";
cin>>b>>h;
area(b,h);
return 0;
}
```

Things we should remember while using default argument are:

a. The default value are assigned to the argument from the right to left convention.

b. Arguments in the functions cannot have default value anywhere if we do not follow right to left convention.

For example:

```
int mul (int i, int j=3, int k=13);     //legal
int mul (int i=5,int j);                //illegal
int mul (int i=0, int j, int k=10);     //illegal
int mul (int i=2, int j=5, int k=10); //legal
```

3.13.5 Pass by Reference

C++ supports one more type of variable called reference variable, in addition to the value variable and pointer variables of C. Value variables are used to hold some numeric values; pointer variables are used to hold the address of some other value variables. Reference variable behaves similar to both, a value variable and a pointer variable. In program code, it is used similar to that of a value variable, but has an action of a pointer variable. Thus, a reference variable provides an alias (alternative name) of the variable that is previously defined. In C, the corresponding parameter in the calling function must be declared as a pointer type. In C++, the corresponding parameter can be declared as any reference type, not just a pointer type.

In pointer variable the returning from the function will be the value while in reference variable in case of returning the value it will return the variable.

3.13.6 Return by Reference

In C++, a function can return a variable by reference. Return by reference means a function is returning an alias of the variable in return statement so the variable which is being return should have a scope, where a function is being invoke. Normally global variable and a variable which is being pass by reference to the function have a scope where

function is being invoke so the return statement returns those variable in return by reference. As a return by reference returns the alias of the variable so the functions call can be place a left side of the assignment operator (=).

Program example:

```
#include<iostream>
using namespace std;
int &max(int&,int&);
int main( )
{
        int a,b;
        cout<<"Enter the number"<<endl;
        cin>>a>>b;
        max(a,b)=-1;
        cout<<"The values are"<<a<<"and"<<b;
return 0;
}
int &max(int &x, int &y)
{
        if(x>y)
          return x;
        else
          return y;
}
```

Ouput:

```
Enter the number
45
63
The value are 45 and -1.
```

Explanation:

Since the return type of max( ) is int&, the function returns reference to x or y (and not the values).

Then function call such as max(a,b) will yield a reference to either a or b depending on their values. This means that this function call can appear on the left hand side of an assignment statement i.e. max(a,b)=-1;

is legal and assigns -1 to a if it is larger, otherwise -1 to b.

3.14 Array, Pointer, and String

Array is the collection of similar data items that can be represented by a single variable name. The individual data items in an array are called elements. A data item is stored in memory in one or more adjacent storage locations depending upon its type. A *pointer* is a special type of variable which holds the address or location of another variable (or data item). Using pointer; the data item is accessed indirectly through its address. In C++, arrays and pointers are closely related. An array name is treated as a pointer constant, and pointers can be subscripted like arrays. String is an array of character. However, string is treated separately than a general array, because they are used in some specific purpose like test manipulation.

3.15 Structure, Union, and Enumeration

Structure

A *structure* is a user defined data type, which contains a collection of different type of variable referenced under single name. The general format for defining syntax is as follows:

```
struct stuct_name
{
  data_type var_name1;
  data_type var_name2;
        .
        .
  data_type var_nameN;
};
```

Union

Union is also a user defined data type. It is same as structure data type i.e., collection of variables of different types. The difference between structure and union is, in structure each member have unique memory location and each member don't effect other members will using it but in unions we can use only one data member at a time, the memory location

of union is shared by all the member of the union. The toal size of structure is the sum of size of individual members but in union the size of largest element in union.

Program example:

```
#include<iostream>
using namespace std;
struct student
{
        public:int var1;
          float var2;
};
union number
{
        public: int var1;
        float var2;
};
int main( )
{
   struct student s1,s2;
   union number s3,s4;
   cout<<"The size of student class is:"<<sizeof(s1)<<endl;
   cout<<"The size of student class variable is:"
   <<&s1.var1 <<"and"<<&s2.var2<<endl;
   cout<<endl<<"The size of number union is:"
   <<sizeof(s3)<<endl;
   cout<<"The size of union variable is:"
   <<&s3.var1<<"and"<<&s4.var2;
return 0;
}
```

Output:

```
The size of student class is:6
The size of student class variable is :0x8f43fff0and0x8f43ffec
The size of number union is:4
The size of union variable is:0x8f43ffe6and0x8f43ffe2
```

Enumerated Data Types

An enumerated data type is a user defined type, with values ranging over a finite set of identifiers called enumeration constants.

E.g., enum color {red, blue, green};

This defines color to be of a new data type which can assume the value red, blue or green. Each of these is an enumeration constant.

E.g., color c;

defines c to be of type color. Internally, the C++ compiler treats an enum type (such as color) as an integer itself.

The above identifiers red, blue, and green represent the integer values of 0,1, and 2 respectively.

cout<<"As an int, c has the value"<<c;

This statement will print

As an int, c has the value 1

We can explicitly assign the constant values to the identifiers red, blue, green. If we assign one value explicitly the corresponding identifier's value will be increase by 1.

E.g., enum color = {red=10, blue, green=34};

If c = blue

cout<<blue; the output will be 11.

Names of different enumeration constant must be distinct. The following example is invalid.

enum emotion {happy, hot, cool};

enum weather {hot, cold, wet};

Because, the name "hot" has the value 1 in the enum "emotion" and the value 0 in enum "weather".

In the above program, if the name hot is used there is ambiguity as to which value to use.

Program example

```
#include<iostream>
using namespace std;
enum days_of_week
{
sun,mon,tue,wed,thrus,fri,sat
```

```
};
int main( )
{
 days_of_week day1,day2;
  day1=sun;
  day2=fri;
  int diff=day2-day1;
  cout<<"Days between="<<diff<<endl;
  return 0;
}
```

Output: Days between=5

SOME EXAMPLES

1. Write a program with function that takes two arguments as reference and assign the average of the two arguments to the smaller one and return that by reference. Call this function by assigning value to the function and display the value of both arguments and call this function without assigning the value and display the value of both the arguments. What will be the output? [2067 Ashad]

```
#include<iostream>
using namepspace std;
float& smaller(float&,float&);
int main( )
{
   float a,b;
   cout<<"Enter two numbers:";
   cin>>a>>b;
   smaller(a,b);
cout<<"Value of a and b without assigning the value:\na=
"<<a<<"\nb="<<b<<endl;
   float avg=(a+b)/2;
   smaller(a,b)=avg;
cout<<"Value of a and b after assigning the value:\na
="<<a<<"\nb="<<b<<endl;
   return 0;
}
float& smaller(float &a,float&b)
{
   if(a<b)
   return a;
        else
   return b;
}
```

2. Use new and delete operators to store *n* numbers dynamically and find their average using casting operator. What are the things we should remember while using default argument. Explain with an example. [2068 Chaitra]

```
#include<iostream>
using namepspace std;
int main( )
{
   int n,i,*arr,tot=0;
   float avg;
   cout<<"How many numbers:";
   cin>>n;
   arr=new int[n];
   cout<<"Enter elements:";
   for(i=0;i<n;i++)
   cin>>arr[i];
   for(i=0;i<n;i++)
  tot+=arr[i];
   avg=static_cast<float>(tot)/n;
   cout<<"Total="<<tot<<endl;
   cout<<"Average="<<avg;
   delete [ ]arr;
   return 0;
}
```

3) Write a meaningful program illustrating the use of function overloading and default arguments. [2069 Ashad]

```
#include<iostream>
using namepspace std;
int simple_interest(int p=100, int t=2, int r=10)
{
  return((p*t*r)/100);
}
int main( )
{
  int principle, time, rate;
  cout<<"Default simple interest is:"
  <<simple_interest( )<<endl;
  cout<<"Enter principle:"<<endl;
  cin>>principle;
  cout<<"One argument simple interest is:" <<simple_interest
  (principle)<<endl;
  cout<<"Enter principle and time:"<<endl;
  cin>>principle>>time;
  cout<<"Two argument simple interest is:" <<simple_interest
  (principle,time)<<endl;
  cout<<"Enter principle, time, and rate:"<<endl;
  cin>>principle>>time>>rate;
  cout<<"Three argument simple interest is :" <<simple_interest
  (principle,time,rate)<<endl;
  return 0;
}
```

4) Write a program to display N number of characters by using default arguments for both parameters. Assume that the function takes two arguments one character to be printed and other number of characters to be printed. [2073 Shrawan]

```
#include<iostream>
using namespace std;
void function(char a='C', int b=10)
{
  cout<<a;
  cout<<endl;
  for(int i=0;i<b;i++)
  cout<<a;
}
int main( )
{
  function( );
  return 0;
}
```

5. With example explain function overloading in object oriented programming. [2075 Ashwin][2075 Chaitra]

```
#include<iostream>
using namepspace std;
int area(int b,int h)
{
        return (0.5*b*h);
}
float area(int r)
{
        return (3.14*r*r);
}
float area(float l,float b)
{
        return (l*b);
}
int main( )
{
        int b,h,r;
        float l,br;
        cout<<"Enter breath and height";
        cin>>b>>h;
        cout<<"Area of Triangle is:"<<area(b,h)<<endl;
        cout<<"Enter radius";
        cin>>r;
        cout<<"Area of Circle is:"<<area(r)<<endl;
        cout<<"Enter length and breath:";
        cin>>l>>br;
        cout<<"Area of rectangle is:"<<area(l,br);
return 0;
}
```

CHAPTER 4:

OBJECTS AND CLASSES

4.1 C++ Classes

A class is a way to bind the data describing an entity and its associated function together. The declaration of a class involves declaration of its four associated attributes:

Data Members are the variables that describe the characteristics of a class. There may be zero or more data members of any type in a class.

Member functions are the set of operations that may be applied to objects of that class. There may be zero or more member functions for a class. They are referred to as the class interface.

Program Access Levels are the visibility of the member of the class within the program. These access levels are private, protected and public.

Class Tagname that is used as specifier using which objects of this class type can be created.

The general form of class definition is as given below:

```
class class_name{
private:
    [variable declarations:]
    [function declarations:]
Protected:
    [variable declarations:]
    [function declarations:]
Public:
    [variable declarations:]
    [function declarations:]
};
```

where the keyword class specifies that it is a class; class_name is the tagname of the class that acts as a type specifier, using which objects of this class type can be created.

There are 3 access specifiers for a class in C++. These access specifiers define how the members of the class can be accessed. Of course, any member of a class is accessible within that class(Inside any member function of that same class). Moving ahead to type of access specifiers, they are:

- **Public** : The members declared as Public are accessible from outside the Class through an object of the class.
- **Protected** : The members declared as Protected are accessible from outside the class **BUT** only in a class derived from it.
- **Private** : These members are only accessible from within the class. No outside Access is allowed.

4.3 Objects and the Member Access

After the declaration of a class, memory is not allocated for the data member of the class. The declaration of class develops a template but data members cannot be manipulated unless an object of its type is created. As with structure we may think that when an object of a class is created memory space is allocated for separate copies of the class's data and member functions. This mental concept emphasizes that the objects are complete, self-contained entities designed using the class declaration. During programming this concept does no harm but assist in understanding abstraction mechanism. However, the reality is quite different. When an object is created, memory is allocated only to its data members but not to member functions.

Defining Member Function

Member functions are created and stored in memory only once when a class specification is declared. All of objects of that class have access to the same area in the memory where the member functions are stored. It is logically true as the member functions are same for all objects and there is no point in allocation a separate copy for each and every object created using the same for class specification. But the separate storage is allocated for every object's data members since they contain different values and allow them to handle their individual data. The space for data is allocated when the object is defined, so there is a separate set of data for each object.

4.4 Defining Member Function

Member functions can be defined in two places:

i. Outside the class definition

ii. Inside the class definition

In both cases, the code for the function body is identical; however, there is a subtle difference.

i. Outside the class Definition

The general form of a member function definition outside the class definition is:

```
return_type class_name::function_name(argument)
{       //function body.
}
```

ii. Inside the class Definition

When a member function is defined inside a class, the function definition is just similar to the function definitions we are familiar with.

```
class class_name
{
        Public:
        return_type function_name( )
{       //function body;
}
};
```

4.5 Constructor

Constructor:

A constructor is a special member function which is called automatically during the object creation. So the constructor can be used for the necessary initialization of the data members. The constructor is identified as a function whose name is same as its class.

The constructor is executed every time an object of that class is defined.

Syntax:

```
class className
{       ...                     //private members
 public:
        className( );           //contructor prototype
};
className::className( )         //constructor definition
{
                                //constructor body definition
}
```

Need of constructors in C++ program:

Prior to C++ programming or Object oriented programming; procedural languages such as C language were used. Procedural language program have typical structure where variables needs to be declared to hold data and functions need to be declared to operate on these variables. To share variables among various parts of program (or with number of functions) either one needs to pass variables as parameters or should declare variable as global.

In every programming scenario, any variable before using it must be initialized to some values. Reason is anything declared but not initialize may contain garbage or junk values. Performing operations on such values may bring our program down.

So, original code author may write some function to initialize these variables or he will initialize at the places where "He" is using them before actually performing some operation. Think about the programmers other than original code author, there are pretty good chances that he will use them without initialising or if there is any function to initialize them, might skip to call it. This will run your program into generating incorrect output, abnormal behaviour or in the worst case program crash.

So, there must be a way to initialize the variables always. Or some initialization function must be called while creating variables. C++ programming language provides a concept called constructors. Constructors guarantee that they will be called as soon as an object of class is created to initialize its member variables. They are named as constructors because they are getting called when an object is constructed.

Example:

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

```
class code
{
      int id;
    public:
      code( ){}              //Default Constructor
      code(int a){id=a;}     //Parameterized Constructor
      code(code &x)          //Copy Constructor
        {
          id=x.id;
        }
      void display( )
        { cout<<id;
        }
};
int main ( )
{
  code A(100);     //object A is created and initialized
  code B(A);       //copy contructor called
  code C=A;        //copy contructor called again
  code D;          //D is created, not initialized
  D=A;             //copy constructor not called
  cout<<"\n id of A:";
  A.display( );
  cout<<"\n id of B:";
  B.display( );
  cout<<"\nid of C:";
  C.display();
  cout<<"\nid of D:";
  D.display( );
  return 0;
  }
```

4.5.1 Default Constructor

A constructor that takes no argument is called a default constructor. A default constructor is automatically called when no arguments are supplied while creating objects as:

class_name object_name;

For example, the default constructor is defined and called as:


```
class counter
{
      private:
            unsigned int count;
      public:
            counter ( )
            {
            }
            ......
};
```

The default constructor counter( ) is called when creating object of the class counter without supplying arguments as:

```
counter c1; //class the default constructor counter( ) and
            //initializes the member data count to zero
```

4.5.2 Parameterized Constructor

It is seen that the default constructor always initializes the object with same value every time they are created. But this may not be the case always. Sometimes it is necessary to create objects with different initial values. The default constructor mechanism does not solve the problem in this case. So we can make constructor that takes arguments and initializes the data members from the values in the argument list. The constructor with parameters are called parameterized constructors. To invoke the parameterized constructor, the agument to the constructor should be passed in parenthesis during object declaration as

```
counter c1(5);
```

For example, the parameterized constructor is created as:

```
class counter
 {
      private:
            unsigned int count;
      public:
            counter ( )
            {     count=0;
            }
            counter (int n) {count=n;}
```

```
        //parameterized constructor initializes the
        //object with the value in the argument.
        //.....
    }
```

4.5.3 Copy Constructor

We use default constructor to initialize the data members to constant value or a parameterized constructor to initialize the data members to values passed as argument. To initialize the object using parameterized constructor we declare the object as:

```
Test t1(5);
```

The object t1 will be initialized by initializing the member with initial value 5. Similarly, we can initialize the object with the object of its own type as:

```
Test t1(5);      //invokes one parameter constructor.
Test t2(t1);     // initialize t2 with the copy of t1 implicitly;
Test t3=t1;      // initialize t3 with the copy of t1 explicitly;
```

The interesting point is that we don't need to create the constructor of this type. The compiler generates a default constructor that will initialize the object with its own type from the value of the object in the argument. This type of constructor that initialize the object by copying the value of the object of its own type from the argument is called copy constructor.

4.6 Destructors

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a function appeared in public section of a class preceded by tilde (~) sign. The destructor never takes any argument nor does it return any values. The constructor is always called to reserve the memory for the instantiated object. The role of destructor is to remove the object from the memory created by constructor.

Syntax:

```
class className
{ ....... //private members
public: ~must be public
        //public members
    ~className( );    //Destructor prototype
};
className::~className( )
{
        //destructor body definition
}
```

4.7 Object as Function Arguments and Return Type

Similar to other variables, the object of class can be passed in both ways:

1. By value
2. By reference

When an object is passed by value, the function creates its own copy of the object and works with its own copy. Therefore, any changes made to the object inside the function do not affect the original object. On the other hand, when an obect is passed by reference, a reference variable is created at function argument so that the claeed function works directly on the original object used in the function call. Thus, any changes made to the object inside the function are reflected in the original object as the function is making changes in the original object itself.

Returing objects from function:

The C++ functions can return any data type to the calling function. The data type can be user defined type like structures or objects. Like the other data types, the functions can return objects to the calling function. Similar to passing objects to the functions as arguments, the objects can also be returned from the function.

Example:

```
#include<iostream>
using namepspace std;
class time
{
   int hr,min,sec;//declaration of hour, minutes and second.
  public:
   void input( )
   {
     cout<<"Enter hour, minutes and seconds";
     cin>>hr>>min>>sec;
```

```
    }
    void display( )
    {
      cout<<hr<<"hour,"<<min<<"Minutes and
      "<<sec<<"Second" <<endl;
    }
    time add(time t1,time t2)
    {
        time temp;
        temp.sec=t1.sec+t2.sec;
        temp.min=t1.min+t2.min+temp.sec/60;
        temp.sec=temp.sec%60;
        temp.hr=t1.hr+t2.hr+temp.min/60;
        temp.min=temp.min%60;
    }

};
int main( )
{
  time t1,t2,t3,t4;
  t1.input( );
  t2.input( );
  t3.add(t1,t2);
  cout<<endl<<"The addition of two time is:";
  t3.display( );
  t5=t4.subtract(t1,t2);
  cout<<endl<<"The subtraction of two time is:";
  t5.display( );
  return 0;
}
```

4.8 Array of Objects

An array of objects is declared after the class definition is over and it is defined in the same way as any other type of array is defined.

Example:

```
class student
{
        char name[20]
        int roll;
public:
        void input( )
        {       cout<<"Enter Name;";
                cin>>name;
                cout<<"Enter roll number:";
                cin>>roll;
        }

};
student e1[10];     //array of object to contain 10 object of
                    //student type
```

4.9 Pointer to Objects and Member Access

Similar to pointer of other data type, we also create pointer type of object of class. Like any other pointer type object also holds the address of another object. The address is assigned using & operator. The general form of declaring pointer type of object is

```
class_name *pointer object;
```

The pointer object to class uses the arrow operator (- >) to access the members of the class. The general form is

```
pointer_object->member_function
```

or

```
(*pointer_object).member function
```

For instance consider, we have Complex class and has input( ) and display ( ) as public members then,

```
Complex *cptr;
Complex C1;
cptr=&C1;
```

Accessing members as:

```
cptr->input( );
cptr->display( );
```

4.10 Dynamic Memory Allocation for Objects and Object Array

In above section, we learned that can create pointer type object of a class like any other type and can be use to access the members of the

class. Similar to other type we can also perform dynamic memory allocation using pointer type object of the class.

```
class_name *pointer_object;
pointer_object=new class_name;
pointer_object=new class_name[size];
```

and to free memory is:

```
delete [ ] pointer_object;
```

Example:

```
#include<iostream.h>
#include<conio.h>
using namespace std;
class person
{private:
     char name[40];
public:
void setname( )
{    cout<<"\nEnter name:";
     cin>>name;
}
void printname( )
{    cout<<"\nName is:"<<name;
}
};
int main( )
{
     person *persptr;
     int n=2,i=0;
     char choice;
     persptr = new person[n];
      do
     {
     persptr[i]->setname( );
     i++;
     }while(i<2);
     for(int j=0;j<n;j++)
     {cout<<"\nPerson number"<<j+1;
               persptr[j]->printname( );
     }
     cout<<endl;
return 0;
}
```

4.11 this pointer

"**this**" is a C++ keyword. "this" always refers to an object that has called the member function currently. We can say that "this" is a pointer. It points to the object that has called this function this time. Always "this" refers to the calling object. "this" can be used in place of the object name. Normally, this pointer is used explicitly in non-static member function when function returns the current object or initializes other object using current object.

Example:

```
#include<iostream>
using namepspace std;
class test
{private:
        int x;
public:
        test (int value)
        { x=value;}
        void print( );
};
void test::print( )
{       cout<<"X="<<x;
        cout<<this;
        cout<<"(*this).x="<<(*this).x;
        cout<<"this->x="<<this->x;
}
int main( )
{
        test t(12);
        t.print( );
return 0;
}
```

Output:

```
X=12
```

```
0x23fefc
(&this).x=12
This->x=12
```

4.12 Static Data Member and Static Function

Static member:

Static member data is not duplicated for each object; rather a single data item is shared by all object of a class. Static variables are normally used in programming to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

```
#include<iostream>
using namespace std;
 class test
 {
      int code;
      static int count;
   public:
     void setcode( )
     {
      code=++count;
     }
     void showcode( )
     {
      cout<<"Object number:"<<code<<"\n";
     }
    static void showcount( )
    {
     cout<<"Count:"<<count<<endl;
    }
};
 int test::count;
int main( )
 {
      test t1,t2;
      t1.setcode( );
      t2.setcode( );
      t1.showcount( );
      t2.showcount( );
      test::showcount( );
      test t3;
      t3.test::setcode( );
      test::showcount( );
      t1.showcode( );
      t2.showcode( );
      t3.showcode( );
      return 0;
 }
```

In above program, showcount( ) is a static member function so it can be called using name of class i.e. test or the object of test class i.e. t1,t2 or t3. As count is static data member and it is initialize to 0. Each time the object of class invokes the setcode( ) function, the count is incremented. So the total number of count is 3. As static data member of class is common to all the objects of class so when we invoke showcode ( ) function it display the different code but the value of count is same i.e. 3.

Static member function:

Static member function is a member function, which can access only the static members of a class. The member function is made static member function by putting keyword static before the function declaration in the class definition.

A static member function is different from other member function in various respects:

a. A static member function can access only static member (function or variable) of the same class.

b. A static member function is invoked by using the class name instead of its objects as

classname::function_name

Why don't you use an object to call the Static Member Function?

Non static data belongs to the object and they can be accessed only through object. Static function can access only static data. They are useful in accessing private static data. So, private static data can be accessed before the object creation through the static function. i.e. Static members

also known as class members are associated with a particular class and not objects so all the objects of the class have same set of static members.

Program that shows the use of static data member and static member function:

```
#include<iostream>
using namespace std;
class test
{
   int code;
   static int count;
  public:
    void setcode( )
    {
     code=++count;
    }
    void showcode( )
    {
     cout<<"Object number:"<<code<<"\n";
    }
    static void showcount( )
    {
     cout<<"Count:"<<count<<endl;
    }
};
int test::count;
int main( )
{
  test t1,t2;
  t1.setcode( );
  t2.setcode( );
  t1.showcount( );
  t2.showcount( );
  test::showcount( );
  test t3;
  t3.test::setcode( );
  test::showcount( );
  t1.showcode( );
  t2.showcode( );
  t3.showcode( );
  return 0;
}
```

In above program, showcount( ) is a static member function so it can be called using name of class i.e. test or the object of test class i.e. t1,t2 or t3. As count is static data member and it is initialize to 0. Each time the object of class invokes the setcode( ) function, the count is incremented. So the total number of count is 3. As static data member of class is common to all the objects of class so when we invoke showcode( ) function it display the different code but the value of count is same i.e. 3.

4.13 Constant Member Functions and Constant Objects.

Const member function:

If a member function of a class does not alter any data in the class, then, this member function may be declared as a constant member functions using the keyword const. For example,

```
void large (int, int) const;
void interest (void) const;
```

The qualifier const appears both in member function declaration and definitions. Once a member function declared as const, it cannot alter the data values of the class. The compiler will generate an error message if such functions try to alter the data values.

Mutable member:

const objects can only invoke const member functions and this const member functions cannot change the data member defined in a class. However, a situation may arise where we want to create const object but we would like to modify a particular data item only. In such situation, we can make it possible by defining the data item as mutable.

The const_cast operator:

There may be situation where we need to modify the constant member in a program. The constant can be object of class or the const function argument of const variable etc. This can be achieved by using const_cast operator which is used to override const or volatile in a cast. The general format is as follows:

const_cast<type>(object);

the destination object type must be a same as that of source type. This attribute allows the const or volatile to be changed.

Program Code:

```
//constant data member and constant function
#include<iostream>
using namespace std;
class ABC
{
  private:
        int ref;
        const int con;
  public:
    ABC(int a, int b) : ref(a),con(b){ }        //initialization list in
                                                //constructor
     void display( )
     {
       cout<<"\nValue of ref:"<<ref;
       cout<<"\nValue of con:"<<con;
     }
};
int main( )
{
    ABC obj(10,12);
    obj.display( );
    return 0;
}
```

Constant Object

C++ allows to define constant objects of user defined classes similar to constants of standard data types. The syntax for defining a constant object is:

```
class_name const object_name(parameter);
        or
const class_name object_name(parameter);
```

- The member of a constant object can be initialized only by a constructor, as a part of object creation procedure

- Once a constant object is created, no member functions of its class can modify its data members. They can only read the contents of the data member. (read only data members).(objects are called read only object)
- The constant member functions are useful for constant object because they guarantee not to change object's value (or data member value)
- Constant function can be invoked by constant as well as non constant object. But, a non constant function can not be called by constant object.

i.e a **constant object can only call constant function**

Example:

```
#include<iostream>
using namespace std;
class Test
{
int data;
public:
Test(int n = 0){data = n;}
void setData(int n){data = n;}
int getData( ) const {return data;}
};
int main( ){
   Test t1(1);
   const Test t2(2);
   t1.setData(5);    //OK
   t1.getData( );    //OK: non const object can call constant function
   t2.setData(7);   //warning: non_const function Test::setData()
                     //called for const object
   t2.getData( );    //OK
   return 0;
}
```

4.14 Friend Function and Friend Classes.

A ***friend function*** is a special type of function in C++ which is not the member function of any classes and that allow us to have access to the private data of those classes.

Any function can be a friend of one class. Any function that is member of another class can also be a friend of one class. If all the member function of a class are to be declared as friend functions then declaring friend for each function requires multiple declarative statements. In such situation, you don't need to make separate friend declarations for all member functions rather you can declare the entire class as the ***friend class***.

Yes, friends violate encapsulation. In general, we know that, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we need to access private members of class by its non-member functions. C++ allows this mechanism through friend function.

Why do we need friend function?

The mechanism of encapsulation and data hiding in C++ do not allow non member functions to access the objects private data. It indicates that if we need to access the private member of the object we must make a public member functions. Sometimes this features leads to considerable inconvenience in programming. If we want to use a function to operate on objects of two different classes, then a function outside a class should be allowed to access and manipulate the private members of the class. In C++, this is achieved by using the concept of friend function.

Example of friend function:

```
#include<iostream>
using namespace std;
class second;                    //forward declaration
class first{
        int data1;
        public:
        first(int x){data1 = x;}
        friend int sum(first, second);
};
class second{
        int data2;
        public:
                second(int x){data2 = x;}
                friend int sum(first, second);
};
int sum(first f, second s){
        return (f.data1 + s.data2);
}
int main( )
{
   first a(15);
   second b(10);
   cout<<"Sum of first and second is: "<<sum(a,b);
   return 0;
}
```

SOME EXAMPLES

1. Write a complete program that illustrates object concept to add and subtract time units (a time unit has hr, min and sec member).

[2066 Jestha]

Answer:

```
#include<iostream>
using namespace std;
class time
{
   int hr,min,sec;   //declaration of hour, minutes and second.
  public:
   void input( )
   {
     cout<<"Enter hour, minutes and seconds";
     cin>>hr>>min>>sec;
   }
   void display( )
   {
     cout<<hr<<"hour,"<<min<<"Minutes and
     "<<sec<<"Second" <<endl;
   }
   void add(time t1,time t2)
   {
     sec=t1.sec+t2.sec;
     min=t1.min+t2.min+sec/60;
     sec=sec%60;
     hr=t1.hr+t2.hr+min/60;
     min=min%60;
   }
```

```
void subtract(time t1, time t2)
{
 min=0;
 hr=0;
 sec=t2.sec-t1.sec;
 if(sec<0)
 {
    sec=sec+60;
    min=min-1;
 }
 min=min+t2.min-t1.min;
 if(min<0)
 {
    min=min+60;
    hr=hr-1;
 }
 hr=hr+t2.hr-t1.hr;
}
};
int main( )
{
 time t1,t2,t3,t4;
 t1.input( );
 t2.input( );
 t3.add(t1,t2);
 cout<<endl<<"The addition of two time is:";
 t3.display( );
 t4.subtract(t1,t2);
 cout<<endl<<"The subtraction of two time is:";
 t4.display( );
 return 0;
}
```

2. Write a program with class called book, which will represent title, author, price, publisher etc. Use constructor for initialization of objects and use dynamic memory allocation for string type member and use destructor in a proper way. [2066 Jestha]

Answer:

```
#include<iostream>
using namespace std;
class book
{
 char *title;
 char *author;
 char *publisher;
 float price;
public:
 book( )
 {
  int length=0;
  title=new char[length+1];
   author=new char[length+1];
   publisher=new char[length+1];
   price=0;
  }
  book (char *t, char *a, char *p,float p1)
  {
   int length;
   length=strlen(t);
    title=new char[length];
    strcpy(title,t);
    length=strlen(a);
    author=new char[length];
    strcpy(author,a);
    length=strlen(p);
    strcpy(publisher,p);
    price=p1;
   }
   book(book &b)
   {
    author=b.author;
     publisher=b.publisher;
     title=b.title;
     price=b.price;
   }
   void display( )
   {
     cout<<"\nTitle of book:"<<title;
     cout<<"\nAuthor of book:"<<author;
```

```
        cout<<"\nPublisher of book:"<<publisher;
        cout<<"\nPrice of book:"<<price;
    }
    ~book( )
    {
     cout<<"\nObject is going to distroyed:";
     delete [ ]author;
     delete [ ]title;
     delete [ ]publisher;
    }
};
int main( )
{
 book b2("C++","Bijarne","India book",550);
 b2.display( );
 book b3(b2);
 b3.display( );
 return 0;
}
```

3. Write a program that will represent time measurement in 12 hour system with object oriented approach. The program should have conversion functions to convert to 12 hour and 24 hour system. [2067 Ashad]

Answer:

```
#include<iostream>
using namespace std;
class time
{
  int hr,min,sec;
 public:
  void getTime( )
  {
    cout<<"Enter hour:";
     cin>>hr;
    cout<<"Enter minutes:";
     cin>>min;
    cout<<"Enter seconds:";
     cin>>sec;
  }
  void convert( )
  {
     if(hr>12)
     hr=hr-12;
  }
  void showTime( )
  {
     cout<<hr<<":"<<min<<":"<<sec<<endl;
  }
};
int main( )
{
  time t1,t2;
  t1.getTime( );
  t2.getTime( );
  t1.convert( );
  t2.convert( );
  t1.showTime( );
  t2.showTime( );
  return 0;
}
```

4. Write a program to create a class LandMeasure that reads Ropani, Ana, Paisa and Dam as data members. Write a function to pass two objects of type Land Measure and return their sum. (16 Ana = 1 Ropani, 4 Paisa=1 Ana, 4 Dam=1 Paisa) [2068 Chaitra]

Answer:

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

```
class LandMeasure
{
    int Ropani,Ana,Paisa,Dam;
  public:
    void input( )
    {
     cout<<"Enter Ropani\n";
     cin>>Ropani;
     cout<<"Enter Ana\n";
     cin>>Ana;
     cout<<"Enter Paisa\n";
     cin>>Paisa;
     cout<<"Enter Dam\n";
     cin>>Dam;
     }
     void display( )
     {
     cout<<"The sum of the LandMeasure is:"<<Ropani
     <<"ropani"<<Ana<<"ana"<<Paisa<<"Paisa"<<Dam<<"dam
";
     }
   LandMeasure sumLandMeasure(LandMeasure, LandMeasure);
};
LandMeasure LandMeasure::sumLandMeasure(LandMeasure L1,
LandMeasure L2)
{
   LandMeasure L3;
   L3.Dam=L1.Dam+L2.Dam;
   if(L3.Dam>4)
   {
      L3.Paisa=L1.Paisa+L2.Paisa+L3.Dam/4;
      L3.Dam=L3.Dam%4;
   }
   if(L3.Paisa>4)
   {
     L3.Ana=L1.Ana+L2.Ana+L3.Paisa/4;
     L3.Paisa=L3.Paisa%4;
   }
   if(L3.Ana>16)
   {
     L3.Ana=L3.Ana%16;
   }
     L3.Ropani=L1.Ropani+L2.Ropani+L3.Ana/16;
   return L3;
}
int main( )
{
      LandMeasure land1,land2,land3;
      land1.input( );
      land2.input( );
      land3=land3.sumLandMeasure(land1,land2);
      land3.display( );
        return 0;
}
```

5) Write a program that can store Department ID and Department name with constructor. Also write destructor in the same class and show that objects are destroyed in reverse order of creation with suitable message. [2068 Chaitra]

Answer:

```
#include<iostream>
#include<string.h>
using namespace std;
class department
{
   int DepartmentID;
```

```
    char DepartmentName[30];
    static int count;
  public:
    department(char *DN,int DI)
    {
      cout<<++count<<" Object Created:"<<endl;
      strcpy(DepartmentName,DN);
      DepartmentID=DI;
      cout<<"Department Name:"<<DepartmentName<<endl;
      cout<<"Department ID:"<<DepartmentID<<endl;
    }
    ~department( )
    {
        cout<<count--<<"Object Destroyed!!"<<endl;
    }
};
int department::count;
int main( )
{
    {
    department D1("Computer",29),D3("Electrical",12);
    }
    {
     department D2("Electronics",19);
    }
  return 0;
}
```

6. Using object oriented technique, write a program to create a class vector that reads integer number. Perform vector addition by passing object as argument and returns the object as result. A vector is a class with array as member. [2069 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class vector
{
      private:
            int x;
            int y;
            int z;
      public:
            void getdata( )
            {
            cout<<"Enter 1st coordinate:";cin>>x;
            cout<<"Enter 2nd coordinate:";cin>>y;
            cout<<"Enter 3rd coordinate:";cin>>z;
            }
            vector addVector(vector A, vector B)
            {
                  vector C;
                  C.x=A.x + B.x;
                  C.y=A.y + B.y;
                  C.z=A.z + B.z;
             return C;
             }
             void array( )
             {
    cout<<"\nX:["<<x<<"]\nY:["<<y<<"]\nZ:["<<z<<"]";
    cout<<endl;
             }
};
int main( )
{
        vector v1,v2,v3;
        v1.getdata( );
        cout<<"First vector:"<<endl;
        v1.array( );
        cout<<endl;
        v2.getdata( );
        cout<<"Second vector:"<<endl;
        v2.array( );
        cout<<endl;
        v3=v3.addVector(v1,v2);
        v3.array( );
      return 0;
}
```

7. Write a program with a class to present time having member functions to read and display time. [2069 Ahad]

Answer:

```
#include<iostream>
using namespace std;
class time
 {
    private:
      int hour,minutes,second;
    public:
      void read( )
      {
       cout<<"Enter the hour, minutes and seconds"<<endl;
       cin>>hour>>minutes>>second;
       }
       void display( )
       {
        cout<<"The given time is :"<<endl;
        cout<<hour<<"hours"<<endl;
        cout<<minutes<<"minutes"<<endl;
        cout<<second<<"seconds";
       }
};
int main( )
{
    time t1;
    t1.read( );
    t1.display( );
    return 0;
}
```

8) Write a program with a class to represent complex numbers. The program should be able to find the sum of two complex numbers. [2069 Ashad]

Answer:

Program code:

```
#include<iostream>
using namespace std;
class complex
{
    int real,imag;
  public:
    complex( )
    { }
    complex(int r, int i)
    {
     real=r; imag=i;
    }
    void addComplex(complex c1,complex c2)
    {
     real=c1.real+c2.real;
     imag=c1.imag+c2.imag;
     }
    void display( )
     {
        cout<<"("<<real<<"+i"<<imag<<")"<<endl;
     }
};
int main( )
{
     complex c1(2,3),c2(4,5),c3;
     cout<<"First complex number:"<<endl;
     c1.display( );
     cout<<"Second complex number:"<<endl;
     c2.display( );
     cout<<"After addition:"<<endl;
     c3.addComplex(c1,c2);
     c3.display( );
     return 0;     }
```

9) Write a program to add members of objects of two different classes. [2070 Ashad]

Answer:

Program Code:

```
#include<iostream>
using namespace std;
class second;      //forward declaration
```

```
class first
{
    int data1;
  public:
    first(int x)
      {data1 = x;}
    friend int sum(first, second);
};
class second
{
    int data2;
  public:
    second(int x){data2 = x;}
    friend int sum(first, second);
};
int sum(first f, second s)
{
  return (f.data1 + s.data2);
}
int main()
{
  first a(15);
  second b(10);
  cout<<"Sum of first and second is:  "<<sum(a,b);
  return 0;
}
```

10) Write a C++ program to join two strings using dynami constructor concept. [2070 Chaitr

Answer:

```
# include <iostream>
# include <string.h>
using namespace std;
class str
{
  char *name;
  int len;
  public:
    str( )
    {
      len=0;
      name=new char[len+1];
    }
    str(char *s)
    {
      len=strlen(s);
      name=new char[len+1];
      strcpy(name,s);
    }
    void show( )
    {
      cout<<"NAME IS:->"<<name<<endl;
    }
    void join(str &a,str &b);
};
void str::join(str &a,str &b)
{
  len=a.len+b.len;
  delete new;
  name=new char[len+1];
  strcpy(name,a.name);
  strcat(name,b.name);
};
int main( )
{
  char *first="Object";
  str n1(first),n2("Oriented"),n3("Programming"),n4,n5;
  n4.join(n1,n2);
  n5.join(n4,n3);
  n1.show( );
  n2.show( );
  n3.show( );
  n4.show( );
  n5.show( );
  getch();
}
```

11. Write a program to find the transpose of given Matrix using the concept of Object Oriented Programming. [2072 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class matrix
{
  int a[3][3];
  int b[3][3];
  //int i1,j1;
public:
  matrix( );
  void transpose( );
  void display( );
};
matrix::matrix( )
{
    for(int i=0;i<3;i++)
    {
      for(int j=0;j<3;j++)
      {
          cin>>a[i][j];
      }
    }
}
void matrix::transpose( )
{
  for(int i=0;i<3;i++)
    {
      for(int j=0;j<3;j++)
      {
        b[i][j]=a[j][i];
      }
    }
}
void matrix::display( )
{
  for(int i=0;i<3;i++)
    {
      for(int j=0;j<3;j++)
      {
        cout<<b[i][j]<<" ";
      }
      cout<<endl;
    }
}
int main( )
{
  matrix m;
  cout<<endl;
  m.transpose( );
  m.display( );
  return 0;
}
```

12. Write a program to create class "time" with data members hours, minute and second. Then add two "time" objects by taking object as argument and also returning object as argument. [2073 Shrawan]

Answer:

```
#include<iostream>
using namespace std;
class time
{
    int hr,minu,sec;//declaration of hour, minutes and second.
  public:
    void input( )
    {
        cout<<"Enter hour, minutes and seconds";
        cin>>hr>>minu>>sec;
    }
    void display( )
    {
        cout<<hr<<"hour,"<<minu<<"Mmnutes and "<<sec<<
        "Second"<<endl;
    }
    time add(time t1,time t2)
```

```
{
    time temp;
    temp.sec=t1.sec+t2.sec;
    temp.minu=t1.minu+t2.minu+temp.sec/60;
    temp.sec=temp.sec%60;
    temp.hr=t1.hr+t2.hr+temp.minu/60;
    temp.minu=temp.minu%60;
    return temp;
  }
};
int main( )
{
    time t1,t2,t3,t4;
    t1.input( );
    t2.input( );
    t4=t3.add(t1,t2);
    cout<<endl<<"The addition of two time is:";
    t4.display( );
    return 0;
}
```

13. Create a class mdistance to store the values in meter and centimeter and class edistance to store values in feet and inches. Perform addition of object of mdistance and object of edistance by using friend function. [2069 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class edistance;    //forward declaration
class mdistance
{
  private:
        float meter;
        float centimeter;
  public:
  mdistance(float m1, float c1)
  {
        meter=m1;
        centimeter=c1;
  }
  friend void addDistance(mdistance one, edistance two);
};
class edistance
{
  private:
        float feet;
        float inches;
  public:
  edistance(float f1, float i1)
  {
        feet=f1;
        inches=i1;
  }
  friend void addDistance(mdistance one, edistance two);
};
void addDistance(mdistance one, edistance two)
{
  float c1=one.centimeter/100;
  float m1=one.meter+c1;
  float i1=two.inches/12;
  float f1=two.feet+i1;
  float met=m1+f1*3.28084;
  int mm=int(met)%100;
  float cc=(met-mm)*100;
  cout<<"Meter: "<<mm<<endl;
  cout<<"Centimeter:"<<cc;
}
int main( )
{
  mdistance m(5,6);
  edistance e(2,4);
  addDistance(m,e);
    return 0;
}
```

14. Create a class called 'time' with data member hour, minute, second and day. Initialize all the data member using constructor. Write a program to add two time object using necessary member function and display the result. [2075 Ashwin]

Answer:

```
#include<iostream>
using namespace std;
class time
{
  int hr,min,sec,day;//declaration of hour, minutes, second and day.
 public:
  time( )
  { day=0;
    hr=0;
    min=0;
    sec=0;
  }

  time(int d,int h, int m, int s)
   {
   day=d;
   hr=h;
   min=m;
   sec=s;
   }
   void display( )
   {

    cout<<day<<"Day"<<hr<<"hour,"<<min<<"Mmnutes and"
          <<sec<<"Second" <<endl;

   }
   void add(time t1,time t2)
   {
    sec=t1.sec+t2.sec;
    min=t1.min+t2.min+sec/60;
    sec=sec%60;
    hr=t1.hr+t2.hr+min/60;
    min=min%60;
    day=t1.day+t2.day+hr/24;
    hr=hr%24;
   }

};
int main( )
{
  time t1(1,12,45,34),t2(3,13,45,34),t3;
  t3.add(t1,t2);
  cout<<endl<<"The addition of two time is:";
  t3.display( );
  return 0;
  }
```

CHAPTER 5:

OPERATOR OVERLOADING

The meaning of the existing operators can be extended to operate on class data. The mechanism of adding special meaning to an operator is called operator overloading. Operator overloading provides a flexible option for the extension of the meaning of the operator when applied to user defined data types.

The operators in C++ such as +, -, /, * can operate on int, float, double etc. But they cannot operate on user defined types such as user's object without extension; that is writing additional piece of code. The meaning of the existing operators can be extended to operate on class data. The mechanism of adding special meaning to an operator is called operator overloading. Operator overloading provides a flexible option for the extension of the meaning of the operator when applied to user defined data types.

5.1 Overloadable Operators

The significance of operator overloading is user-defined data types behave like built – in data types, thus allows the user to extend the language and makes the code more readable. That is operator overloading allows redefining additional meaning to operators.

C++ supports the operator overloading, but at least operand used with operator should be the instance of class i.e. object of class. Following are the overloadable operators.

Operators	Description
()	Function Call
[ ]	Array element reference
++, --	Unary operator
+, -, *, /, %	Arithmetic operators
&,|,~,^,<<,>>	Bitwise
&=,|=,^=,<<=,>>=	Bitwise assignment
+=,-=,*=,/=,%=	Arithmetic assignment
&&,||,!	Logical operators
->	Member access through

Operators	Description
->*	Member access through member pointer
new,new[],delete,delete[]	Dynamic Memory allocation and release
*	Dereference operator
,	Comma operator
=	Assignment
<,>,==,!=,<=,>=	Relational operator

We can overload all the C++ operators except the following:

a. Class member access operator (., .*).
b. Scope resolution operator (::).
c. Size of operator (**sizeof**).
d. Conditional operator (? :).

In binary operator overloading using member function right-hand operand is supplied as argument to the operator function, the right-hand operand can be object of same class or different or the fundamental type depending upon situation. But the left-handed operand must be the object of class where operator function is defined and left-hand operand is supplied implicitly to operator function as it is same as object invoking the member function.

5.2 Syntax of Operator Overloading

Syntax of operator overloading:

```
classclass_name
{       .....
        public:
friend return_type operator operator_symbol ([arg, [arg]]);
        .......
};
return_type operator operator_symbol([arg,[arg]])
{
        //body of function
}
```

5.3 Rules of Operator Overloading

Rules of operator overloading:

a. Only existing operators can be overloaded. New operators cannot be created.

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

c. 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.

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

e. 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 relevant class).

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

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

5.4 Unary Operator Overloading

The operators which operate on a single operand (data) are called unary operator. The unary operators in C++ are either prefix or postfix with the operand. The unary operator either prefix or postfix can be overloaded with non static member function taking no argument or a non member function (usually global function) taking one argument. After overloading the operator, it can be applied to an object in the same way as it is applied to the basic types.

Syntax:

```
class ,class_name
{
  public:
        return_type operator_symbol ( );          //prefix
        return_type operator_symbol(int);         //postfix
};
//for prefix
return_type class_name::operator_symbol ( )
{//.....}
return_type class_name:: operator_symbol (int)
{//.....}
```

Example:

```
#include<iostream>
using namespace std;
class Counter
{
   private:
     unsigned int count;
   public:
     Counter( )
     {
        count=0;
     }
     int get_count( )
     {
        return count;
     }
     void operator++(int)
     {
        count++;
     }
     void operator++( )
     {
        count++;
     }
};
void main( )
{
     Counter c1,c2;
     cout<<"\nc1="<<c1.get_count();
     cout<<"\nc2="<<c2.get_count();
     c1++;
     c2++;
     ++c2;
     cout<<"\nc1="<<c1.get_count();
     cout<<"\nc2="<<c2.get_count();
     return 0;
}
```

5.5 Binary Operator Overloading

The operators which operate on two operands (data) are called binary operators. The binary operator function can be defined by either a non static member function taking one argument or a non member function (usually global function) taking two arguments. After overloading any operator it can be applied to an object in the same way as it is applied to the basic type.

Syntax:

```
class class_name
{
  public:
        return_type operator_symbol ( class_name arg );
};
return_type class_name::operator_symbol ( class_name arg )
{//.....}
```

The binary operator can also be defined as a non member function of the class. The binary operator defined as a non member function has the following form

```
return_type class_name::operator_symbol ( class_name obj1, class_name obj2 )
{//.....}
```

Example

```
#include<iostream>
using namespace std;
class complex
{
   private;
     float real;//real part of complex number.
     float imag;//imag. part of complex number.
   public:
     complex( ) //no argument constructor.
     {
      real=imag=0.0;
     }
     complex(float x,float y)
     {
      real=x;
      imag=y;
     }
     complex operator+(complex);  //Add complex
     void outdata(char *msg)
     {
        cout<<endl<<msg;
        cout<<"("<<real;
        cout<<","<<imag<<")";
     }
};
//adds default and C2 complex objects.
complex complex::operator+(complex c2)
{
     complex temp;        //object temp of complex class
     temp.real=real+c2.real;        //object temp of complex class
     temp.imag=imag+c2.imag;    //add imaginary parts
     return temp;
}
int main( )
{
     complex c1(2.5,3.5),c2(1.6,2.7),c3;    //c1,c2,c3 are object
                                             //of complex class
     c1.outdata("C1=");
     c2.outdata("C2=");
     c3=c1+c2;
     c3.outdata("c3=c1+c2: ");
     return 0;
}
```

5.6 Operator Overloading with Member and Non Member Functions

Overloading operators using member function:

```
returnType operator operator_symbol (argu_list)
{
  //body of function.
}
```

Overloading operator using nonmember function:

```
friend returnType operator operator_symbol (argu_list)
{
  //body of function.
}
```

Disadvantage of using operator overloading in C++

The encapsulation feature is violated in object oriented method if we use non member function operator overloading i.e. using friend function in operator overloading. This is the disadvantage in operator overloading since it violates the data hiding concept.

5.7 Data Conversion: Basic-User Defined and User Defined – User Defined.

Data conversion:

The = operator will assign a value from one variable to another, in statements like

```
int var1=int var2;
```

where int var1 and int var2 are integer variables. We may also have noticed that = assigns the value of one user-defined object to another, provided they are of the same type, in statements like

```
dist3 = dist1 + dist2;
```

where the result of the addition, which is type Distance, is assigned to another object of type Distance, dist3. Normally, when the value of one object is assigned to another of the same type, the values of the all the member data items are simply copied into the new object. The complier doesn't need any special instructions to use = for the assignment of user-defined objects such as Distance objects.

What of the assignment operator is used for different type i.e. float to int or int to user defined etc. For these conversions, compiler will not perform it but users need to tell it to perform. In the following section we will deal about how to hand the data conversion.

CONVERSION FROM BASIC TO USER-DEFINED TYPE

```
#include<iostream>
using namespace std;
class Meter
{
private:
  float length;
public:
  Meter( )
   {  length=0; }
  Meter(float l)
   {  length=l/100.0; }
  void showlength( )
      {
        cout<<"Length(in meter)="<<length;
      }
};
int main( )
{
        Meter m1;
        float l1;
        cout<<"Enter length(in cms):";
        cin>>l1;
        //m1 is user-defined and l1 is basic
        m1=l1;  //convert from basic to user defined
        m1.showlength();
        return 0;
}
```

CONVERSION FROM USER-DEFINED TO BASIC

```
#include<iostream>
using namespace std;
class Meter
{
    private:
      float length;
    public:
      Meter( )
        {
        length=0;
        }
      operator float( )
        {
        float l;
```

```
        l=length*100.0;
        return (l);
        }
      void getlength( )
        {
        cout<<"\nEnter length (in meters):";
        cin>>length;
        }
};
int main( )
{
        Meter m1;
        float l1;
        m1.getlength( );
        //m1 is user-defined and l1 is basic
        l1=m1;//convert from user-defined to basic
        cout<<"Length in cms="<<l1;
        return 0;
}
```

CONVERSION FROM USER-DEFINED TO USER-DEFINED

I. Conversion Routine in Source object: operator function

```
#include<iostream>
#define pi 3.14159
using namespace std;
class Radian
{
  private:
    float rad;
  public:
    Radian( )
    {
      rad=0.0;
    }
    Radian(float r)
    {
      rad=r;
    }
    float getradian( )
    {
      return rad;
    }
    void display( )
    {
      cout<<"Radian="<<getradian();
    }
};
class Degree
{
  private:
    float degree;
  public:
    Degree( )
      {
      degree=0.0;
      }
    operator Radian( )
      {
      float radian;
      radian=degree*pi/180.0;
      return(Radian(radian));
      }
    void input( )
      {
      cout<<"Enter degree";
      cin>>degree;
      }
};
int main( )
{
      Degree d1;
      Radian r1;
      //d1 and r1 are objects
      d1.input( );
      r1=d1;
      r1.display( );
      return 0;
}
```

II. Conversion Routine in destination object: constructor function

```
#include<iostream>
#define pi 3.14159
using namespace std;
class Degree
{
    private:
      float degree;
    public:
      Degree( )
        {
        degree=0.0;
        }
      float getdegree( )
        {
        return degree;
        }
      void input( )
        {
        cout<<"Enter degree";
        cin>>degree;
        }
};
class Radian
{
    private:
      float rad;
    public:
      Radian( )
        {
        rad=0.0;
        }
      float getradian( )
        {
        return rad;
        }
      Radian(Degree deg)
        {
        rad=deg.getdegree()*pi/180.0;
        }
      void display( )
        {
        cout<<"Radian="<<getradian();
        }
};
int main( )
{
    Degree d1;
    Radian r1;
    //d1 and r1 are objects
    d1.input();
    r1=d1;
    r1.display();
    return 0;
}
```

5.8 Explicit Constructor.

There may be situation where you want a specific conversion to take place, as you studied in previous sections, which can be done using constructor with one argument, and using casting operator function. However, there may be situation where you don't want these conversions to take place.

It is easy to prevent conversion using casting function, just don't define casting function inside the class. However, preventing through constructor is not as easy as you may need one argument constructor to initialize the data member of class. To prevent this implicit conversion, ANSI C++ standard have define a keyword explicit. The keyword 'explicit' is placed just before the declaration of a one-argument constructor to declare constructor as 'explicit' constructor. Consider the following code fragment.

```
class XYZ
{
  int A;
  public:
        explicit XYZ(int m)
        {
```

```
            A=m;
        }
            //other member of class
};
```

When we declare object of XYZ as below:

```
XYZ obj(5);     //object can be created.
```

And

```
XYZ Obj1=45;
```

Is not allowed and illegal.

SOME EXAMPLES

1. Write a program that will add objects of a data class (with members day, month and year) using operator overloading. Make another function to find no of days in between two dates by overloading operator. [2066 Jestha]

Answer:

```
#include<iostream>
using namespace std;
class date
{
   int year,month,day;
  public:
    void getdata( )
    {
    cout<<"\nEnter the year, month and day:";
    cin>>year>>month>>day;
    }
    void display( )
    {
    cout<<endl<<"Year:"<<year<<endl<<"Month:"<<month<<"
        \nDay:"<<day;
    }
    date operator+(date d)
    {
     date temp;
     temp.day=day+d.day;
     temp.month=month+d.month+temp.day/30;
     temp.day=temp.day%30;
     temp.year=year+d.year+temp.month/12;
     temp.month=temp.month%12;
     return temp;
    }
friend int operator-(date,date);
};
int operator-(date d1,date d2)
{
  int total=0;
  total=d1.day-d2.day;
  total=(d1.month-d2.month)*12+total;
  total=(d1.year-d2.year)*365+total;
  return total;
}
int main( )
{
  date d1,d2,d3;
  int day;
  cout<<"\nEnter the start date:";
  d1.getdata( );
  cout<<"\nEnter the stop date:";
  d2.getdata( );
  d3=d2+d1;
  cout<<"\nThe sum of two date is:";
  d3.display( );
  day=d2-d1;
```

```
    cout<<"\nEnter the days between two date is:"<<day;
    return 0;
}
```

2. Write a program to convert object of a class that represents weight of gold in tola to object of class that represent weight in grams. (1 tola = 11.664gm) [2067 Ashad]

Answer:

Conversion Routine in Source object: operator function [Converting tola to gold]

```
#include<iostream>
using namespace std;
class gram
{
    private:
        float wt1;
    public:
        gram( )
        {
         wt1=0.0;
        }
        gram(float g)
        {
         wt1=g;
        }
        float getgram( )
        {
          return wt1;
        }
        void display( )
        {
          cout<<"Gram="<<getgram();
        }
};
class tola
{
    private:
        float wt2;
    public:
        tola( )
        {
         wt2=0.0;
        }
        operator gram( )
        {
          float g1;
          g1=wt2*11.664;
          return(gram(g1));
        }
        void input( )
        {
         cout<<"Enter tola";
         cin>>wt2;
        }
};
int main( )
{
        tola t1;
        gram g1;
        //t1 and g1 are objects
        t1.input( );
        g1=t1;
        g1.display( );
        return 0;
}
```

3. Create a class Complex with two member variables real and imaginary of type float. Write default, parameterized and copy constructors. Make necessary function to display the state of the object on the screen. Overload arithmetic assignment operator "+=" and stream operator ">>". Write main function to test the class. [2067 Magh]

Answer:

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

```
class complex
{
   private:
     float real;
     float imag;
   public:
     complex( )          //default constuctor
      {
        real=0.0;
        imag=0.0;
      }
      complex(float r, float i)    //parameterized constructor
      {
        real=r;
        imag=i;
      }
      complex(complex &c)  //copy constructor
      {
        real=c.real;
        imag=c.imag;
      }
      void display( )
      {
        cout<<"\n("<<real<<"+i"<<imag<<")\n";
      }
};
int main( )
{
        complex c1(2,3);  /*parameterized call*/
        complex c3;       /*default constructor call;*/
        cout<<"Initial value of C1 object is:";
        c1.display( );
```



```
        cout<<"Initial value of c3 object is:";
        c3.display( );
        cout<<"copying data of c1 object to c2:";
        complex c2(c1);//copy constructor call
        c2.display( );
        return 0;
}
```

Program code: Overloading += operator

```
#include<iostream>
using namespace std;
class complex
{
  private:
    float real;//real part of complex number.
    float imag;//imag. part of complex number.
  public:
    complex( ) //no argument constructor.
  {real=imag=0.0;}
    complex(float x,float y)
  {
          real=x;
          imag=y;
  }
    void operator+=(complex);//Add complex
    void outdata(char *msg)
     {
  cout<<endl<<msg;
  cout<<"("<<real;
  cout<<","<<imag<<")";
     }
};
//adds default and C2 complex objects.
```

```
void complex::operator+=(complex c2)
{
  real+=c2.real;    //object temp of complex class
  imag+=c2.imag; //add imaginary parts
}
int main( )
{
        complex c1(2.5,3.5),c2(1.6,2.7);   //c1,c2,c3 are object
                                           //of complex class
  c1.outdata("C1=");
  c2.outdata("C2=");
  c1+=c2;
  c1.outdata("c3=c1+c2: ");
        return 0;
}
```

Program code: Overloading >> operator

```
#include<iostream>
using namespace std;
class complex
{
  int real;
  intimag;
  public:
  friend istream& operator>>(istream&, complex &);
  friend ostream& operator<<(ostream&, complex &);
};
istream& operator>>(istream&input,complex&c)
{
    cout<<"\nEnter the real part:";
  input>>c.real;
  cout<<"\nEnter the imaginary part:";
  input>>c.imag;
    return input;
}
ostream& operator<<(ostream&output,complex&c)
{
    output<<endl<<c.real<<"+i"<<c.imag;
    return output;
}
int main( )
{
  complex c1;
  cin>>c1;
  cout<<"\nThe complex number is :";
  cout<<c1;
    return 0;
}
```

4. Write a program to compare the magnitude of complex number by overloading <, > and ==operator. [2068 Baisakh]

Answer:

```
#include <iostream>
#include <math.h>
using namespace std;
class complex
{
      int real;
      int imag;
   public:
      void getdata( )
      {
        cout<<"Enter real value";
        cin>>real;
        cout<<"Enter imaginary value";
        cin>>imag;
      }
      float operator<(complex c)
```

```
{
float mag1=sqrt(real*real+imag*imag);
float mag2=sqrt(c.real*c.real+c.imag*c.imag);
return (mag1<mag2)?true:false;
 }
 float operator>(complex c)
 {
    float mag1=sqrt(real*real+imag*imag);
    float mag2=sqrt(c.real*c.real+c.imag*c.imag);
    return (mag1>mag2)?true:false;
  }
  float operator=(complex c)
  {
     float mag1=sqrt(real*real+imag*imag);
     float mag2=sqrt(c.real*c.real+c.imag*c.imag);
     return (mag1==mag2)?true:false;
    }
    void display( )
    {
       if(imag<0)
       cout<<"\n"<<real<<"-j"<<(imag*-1);
       else
       cout<<"\n"<<real<<"+j"<<imag;
     }
};
int main( )
 {
    complex d1,d2;
    d1.getdata( );
    d2.getdata( );
    d1.display( );
    cout<<endl;
```



```
    d2.display( );
    if(d1<d2)
    cout<<endl<<"First complex number is smaller";
    else if(d1>d2)
    cout<<endl<<"First complex number is greater";
    else if (d1=d2)
    cout<<endl<<"Both complex number are equal";
    return 0;
}
```

5. Create a class having an array as member. Overloaded index operator ([ ]) to input and display the elements in the array. **[2068 Chaitra]**

Answer:

```
#include<iostream>
#include<process.h>
using namespace std;
constint size=4;
class Array
{
        int a[size];
  public:
     Array(int s);
     int operator[](int index)
     {
        if(index<0||index>size)
        {
        cout<<"\n Bound exception:";
       exit(0);
       }
       return a[index];
     }
};
```

```
Array::Array(int s)
   {
        for(int i=0;i<s;i++)
        cin>>a[i];
   }
int main( )
{
int n;
cout<<"Enter the number of array element to display:"<<endl;
cin>>n;
Array A(n);
for(int i=0;i<n;i++)
cout<<"\n a["<<i<<"] element is:"<<A[i];
return 0;
}
```

6. Write a program to multiply two matrices using operator overloading.

Answer:

```
#include<iostream>
using namespace std;
class matrix
{
   int a[3][3];
  public:
    matrix( );//default constructor
    void set( );// to set matrix elements
    void show( );// to show matrix elements
    /*
binary * operator will require one more arg, since it is a binary
operator
onearg is the object itself that will call it, other will be passed as
arg(in this case x). Also this will return a matrix object
  */
    matrix operator*(matrix x);  // overloading * for multiplication
};
```

```
matrix::matrix( )   //default constructor
{
 for(int i=0;i<3;i++)
 {
  for(int j=0;j<3;j++)
  {
   a[i][j]=0;
  }
 }
}
void matrix::set( )  // to set matrix elements
{
 for(inti=0;i<3;i++)
 {
  for(int j=0;j<3;j++)
  {
   cout<<"\n Enter "<<i<<","<<j<<" element=";
   cin>>a[i][j];
  }
 }
}
void matrix::show( )  // to show matrix elements
{
 cout<<"\n Matrix is=\n";
 for(int i=0;i<3;i++)
 {
  for(int j=0;j<3;j++)
  {
   cout<<a[i][j]<<",";
  }
  cout<<"\n";
 }
}
matrix matrix::operator*(matrix x) // overloading * for
                                   //multiplication
{
 matrix c; // this will hold our result
 for(int i=0;i<3;i++)
 {
```

```
        for(int j=0;j<3;j++)
        {
          c.a[i][j]=0;
          for(int k=0;k<3;k++)
          {
            c.a[i][j]=c.a[i][j]+a[i][k]*x.a[k][j];
          }
          }
      }
      return(c);
    }
  int main( )
  {
     matrixa,b,c;
     a.set( );
     b.set( );
     c=a*b;
  /*
  note that compiler will break this statement as
    c=a.operator*(b);
  this is how 2nd arg is passed.
  and this is how object "a" acts as the calling object
    */
     a.show( );
     b.show( );
     c.show( );
     return 0;
}
```

7. Write a program to add two matrices by overloading the + operator. [2070 Ashad]

Answer:

```
#include<iostream>
using namespace std;
class matrix
{
   int a[3][3];
  public:
   matrix( );      //default constructor
   void set( );    // to set matrix elements
   void show( );   // to show matrix elements
   matrix operator+(matrix x);     // overloading + for addition
};
matrix::matrix( )  //default constructor
{
  for(int i=0;i<3;i++)
  {
    for(int j=0;j<3;j++)
      {
       a[i][j]=0;
      }
  }
}
void matrix::set( ) // to set matrix elements
{
  for(int i=0;i<3;i++)
  {
    for(int j=0;j<3;j++)
    {
      cout<<"\n Enter "<<i<<","<<j<<" element=";
      cin>>a[i][j];
    }
  }
}
void matrix::show( ) // to show matrix elements
{
  cout<<"\n Matrix is=\n";
  for(int i=0;i<3;i++)
  {
    for(int j=0;j<3;j++)
    {
      cout<<a[i][j]<<",";
    }
    cout<<"\n";
  }
}
matrix matrix::operator+(matrix x)  // overloading * for
                                    //multiplication
```

```
{
 matrix c;// this will hold our result
 for(int i=0;i<3;i++)
  {
   for(int j=0;j<3;j++)
    {
       c.a[i][j]=a[i][j]+x.a[i][j];
    }
  }
  return(c);
 }
 int main( )
 {
   matrix a,b,c;
   a.set( );
   b.set( );
    c=a+b;
   a.show( );
   b.show( );
   c.show( );
   return 0;
 }
```

8. Write a program to define a Class Distance with necessary data members and functions. Then overload the relational operators to compare the two objects of Distance class. [2070 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class distance
 {
      long int feet,inches;
    public:
      distance( )
      {
       feet=0;
       inches=0;
       }
       void read( )
{
 cout<<"Enter value in Feet and inches";
 cin>>feet>>inches;
 inches+=feet*12;
}
distance operator+(distance a)
 {
   distance temp;
  temp.inches=inches+a.inches;
    temp.feet=temp.inches/12;
    temp.inches=temp.inches%12;
    return temp;
 }
 distance operator-(distance a)
  {
     distance temp;
   temp.inches=inches-a.inches;
   temp.feet=temp.inches/12;
   temp.feet=temp.inches%12;
   return temp;
   }
   void display( )
   {
    cout<<feet<<"Feet and "<<inches<<"inches"<<endl;
   }
  int operator>(distance a)
   {
 if (inches>a.inches)
 return 1;
 else
 return 0;
    }
    int operator<(distance a)
     {
  if (inches<a.inches)
  return 1;
  else
  return 0;
     }
```

```
    int operator==(distance a)
      {
    if (inches==a.inches)
    return 1;
    else
    return 0;
      }
     int operator != (distance a)
      {
    if (inches != a.inches)
    return 1;
    else
    return 0;
      }
  };
int main( )
  {
     distance c1,c2,c3,c4;
     c1.read( );
     c2.read( );
     c3=c1+c2;
     cout<<"Addition:";
     c3.display( );
     c4=c1-c2;
     cout<<"Subtraction:";
     c4.display( );
     if (c1==c2)
     cout<<"The values are equal.";
     if (c1 != c2)
     cout<"The values are different and ";
     if(c1>c2)
     cout<<"1st value is greater one.";
     if (c1<c2)
     cout<<"2nd value is greater one.";
     return 0;
```

9. Write a program that will convert object from a class Rectangle to object of a class Polar using Casting Operator. [2072 Chaitra]

Answer:

```
#include<iostream>
#include<cmath>
using namespace std;
class Polar
{
    float radius;
    float theta;
  public:
    Polar( )
     {
     radius=0;
     theta=0;
     }
    Polar(float rad,float th)
     {
     radius=rad;
     theta=th;
     }
    void display( )
     {
     cout<<"("<<radius<<","<<theta<<")";
     }
};
class Rectangle
{
    float x;
    float y;
   public:
    Rectangle ( )
     {
     x=0;
     y=0;
     }
```

```
    Rectangle(float x1,float y1)
    {
    x=x1;
    y=y1;
    }
    operator Polar ( )
    {
    float r=static_cast<float>(sqrt(x*x+y*y));
    float t=static_cast<float>(atan(y/x));
    return Polar(r,t);
    }
    void display( )
    {
      cout<<"("<<x<<","<<y<<")";
    }
};
int main( )
{
      Rectangle rec(12,5);
      Polar pol;
      pol=rec;
      cout<<"Given rectangle:";
      rec.display( );
      cout<<"\nEquivalent Polar";
      pol.display( );
      return 0;
}
```

10. Create a class named City that will have two member variables CityName(char[20]) and DistFromKtm(float). Add member functions to set and retrieve the CityName and DistFromKtm separately. Add operator overloading to find the distance between the cities (just find the difference of DistFromKtm) and sum of distance of those cities from Kathmandu. In the main function, initialize three city objects. Set the first and second city to be Pokhara and Dhangadi. Display the sum of DistFromKtm of pokhara and Dhangadi and distance between Pokhara and Dhangadi.

Answer:

```
#include<iostream>
#include<string.h>
using namespace std;
class City
{
      char CityName[20];
      float DistFromKtm;
    public:
      void setCity(char *cn,float dfk)
      {
       strcpy(CityName,cn);
       DistFromKtm=dfk;
      }
      float operator-(City c1)
      {
       return(this->DistFromKtm-c1.DistFromKtm);
      }
      float operator+(City c1)
      {
       return(this->DistFromKtm+c1.DistFromKtm);
      }
      void display( )
      {
       cout<<"CityName:"<<CityName<<endl;
cout<<"Distance from Kathmandu:"<<DistFromKtm<<endl;
      }
};
int main( )
{
  City Pokhara,Dhangadi;
  Pokhara.setCity("Pokhara",206);
  Dhangadi.setCity("Dhangadi",671);
  Pokhara.display( );
  cout<<endl;
  Dhangadi.display( );
  cout<<endl;
```

```
        cout<<"Sum of Distance of Kathmandu of Pokhara and
              Dhangadi is:"<<Dhangadi+Pokhara<<endl;
        cout<<"Distance between Pokhara and Dhangadi is:"
              <<Dhangadi- Pokhara;
        return 0;
    }
```

11. Write a program to overload relational operators (==, !=, >, <, >=, <=) to compare complex numbers. [2069 Chaitra]

Answer:

```
#include <iostream>
#include <math.h>
using namespace std;
class complex
{
  int real;
  int imag;
  public:
  void getdata( )
   {
   cout<<"Enter real value";
          cin>>real;
         cout<<"Enter imaginary value";
         cin>>imag;
  }
  float operator<(complex c)
  {
    return (real<c.real || imag<c.imag)?true:false;
  }
  float operator>(complex c)
  {
    return (real>c.real || imag>c.imag)?true:false;
  }
  float operator==(complex c)
  {
    return (real==c.real || imag==c.imag)?true:false;
  }
  float operator!=(complex c)
  {
    return (real!=c.real || imag!=c.imag)?true:false;
  }
  float operator<=(complex c)
  {
    return (real<=c.real || imag<=c.imag)?true:false;
  }
  float operator>=(complex c)
  {
    return (real>=c.real || imag>=c.imag)?true:false;
  }
  void display( )
  {
  if(imag<0)
          cout<<"\n"<<real<<"-j"<<(imag*-1);
    else
          cout<<"\n"<<real<<"+j"<<imag;
  }
};
int main( )
{
  complex d1,d2;
  d1.getdata( );
  d2.getdata( );
  d1.display( );
  cout<<endl;
  d2.display( );
   if(d1<d2)
  cout<<endl<<"First complex number is smaller";
  if(d1>d2)
  cout<<endl<<"First complex number is greater";
  if (d1!=d2)
  cout<<endl<<"Both complex number are unequal";
  if (d1<=d2)
  cout<<endl<<"Both complex number are equal and smaller";
   if (d1=d2)
   cout<<endl<<"Both complex number are equal";
return 0;
}
```

(3,9) , (5,8)

12. Write a program to overload the relational operators to compare the length (in meter and centimeter) of two objects. [2074 Ashwin]

Answer:

```
#include<iostream>
using namespace std;
class distance1
{
   long int meter,centimeter;
public:
   distance1( )
   {
  meter=0;
  centimeter=0;
   }
   void read( )
   {
  cout<<"Enter value in Meter and Centimeter";
  cin>>meter>>centimeter;
  centimeter+=meter*100;
   }
   distance1 operator+(distance1 a)
   {
  distance1 temp;
  temp.centimeter=centimeter+a.centimeter;
  temp.meter=temp.centimeter/100;
  temp.centimeter=temp.centimeter%100;
  return temp;
   }
   distance1 operator-(distance1 a)
   {
  distance1 temp;
  temp.centimeter=centimeter-a.centimeter;
  temp.meter=temp.centimeter/100;
  temp.centimeter=temp.centimeter%100;
  return temp;
   }
   void display()
   {
cout<<meter<<"Meter and "<<centimeter<<"centimeter"<<endl;
   }
   int operator>(distance1 a)
   {
  if (centimeter>a.centimeter)
     return 1;
  else
     return 0;
   }
   int operator<(distance1 a)
   {
  if (centimeter<a.centimeter)
     return 1;
  else
     return 0;
   }
   int operator==(distance1 a)
   {
  if (centimeter==a.centimeter)
     return 1;
  else
     return 0;
   }
   int operator != (distance1 a)
   {
  if (centimeter != a.centimeter)
     return 1;
  else
     return 0;
   }
};
int main( )
{
   distance1 c1,c2,c3,c4;
   c1.read( );
   c2.read( );
   c3=c1+c2;
   cout<<"Addition:";
   c3.display( );
```

```
c4=c1-c2;
cout<<"Subtraction:";
c4.display( );
if (c1==c2)
cout<<"The values are equal.";
 if (c1 != c2)
cout<"The values are different and ";
 if(c1>c2)
cout<<"1st value is greater one.";
 if (c1<c2)
cout<<"2nd value is greater one.";

 return 0;
}
```

13. Write operator functions as member function of a class to overload arithmetic operator '+', logical operator '<=' and stream operator '<<' to operate on the objects of user defined type time (hr, min, sec). [2074 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class time
{
   int hr, min, sec;
public:
   friend istream & operator>>(istream &, time &);
   friend ostream & operator<<(ostream &, time &);
   time operator+(time);
   float operator<=(time t);
};
istream & operator>>(istream &input,time &t)
{ cout<<"\nEnter Hour:";
          input>>t.hr;
          cout<<"\nEnter Minutes:";
          input>>t.min;
          cout<<"\nEnter Second:";
          input>>t.sec;
    return input;
}
ostream & operator<<(ostream &output,time &t)
{ output<<endl<<t.hr<<":"<<t.min<<":"<<t.sec;
  return output;
}

time time::operator+(time t3)
{
   time temp;
  temp.sec=sec+t3.sec;
  temp.min=min+t3.min+temp.sec/60;
  temp.sec=temp.sec%60;
   temp.hr=hr+t3.hr+temp.min/60;
   temp.min=temp.min%60;
   return temp;

}
float time::operator <=(time t)
{
   float first=sec+min*60+hr*60*60;
   float second=t.sec+t.min*60+hr*60*60;
   return(first<=second)?true:false;
}
int main( )
{
   time t1, t2, t3;
   cin>>t1;
   cin>>t2;
   if(t1<=t2)
      cout<<"First time is less or equal to second time;"<<endl;
   else
      cout<<"First time is greater than second time:"<<endl;
   t3=t1+t2;
   cout<<"After Addition:"<<endl;
   cout<<t3;
   return 0;
}
```

14. Write a program to concatenate two user given string by overloading binary plus (+) operator. [2075 Ashwin]

Answer:

```
#include<iostream>
#include<string.h>
using namespace std;
class strings
{
  char a[30];
public:
  void read( )
  {
    cout<<"Enter string:";
    cin>>a;
  }
  void display( )
  {
    cout<<"The string is:"<<a;
  }
  strings operator+(strings b)
  {
    strings temp;
    strcpy(temp.a,strcat(a,b.a));
    return temp;
  }
};
int main( )
{
  strings s1,s2,s3;
  s1.read( );
  s2.read( );
  s3=s1+s2;
  cout<<"After concatenation:";
  s3.display( );
  return 0;
}
```

CHAPTER 6: INHERITANCE

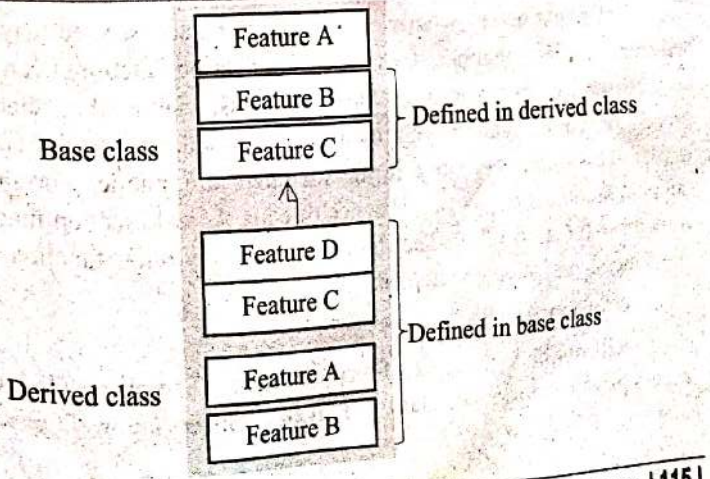
Derivation of Inheritance:

The derivation of inheritance refers to the length of its path from the root (top base class). A base class itself might have been derived from another classes in the class hierarchy. During this level of inheritance different members are created in derived classes hence different error like ambiguity, type mismatch might be commited.

Inheritance is a process of organizing information in a hierarchical form. Through inheritance we can create new class, called derived classes, from existing class called base class. The derived class inherits all the features of the base class and can add new extra features. Inheritance allows new classes to be built from existing and less specialized class instead of writing from the scratch. So classes are created by first inheriting all the features from base class and adding new features in the derived class.

Inheritance is an essential part of object oriented programming. Inheritance provide code reusability. Reusability is one of the striking features of C++. We don't have to write code from the scratch each and every time while writing similar type of construct. The reused code is of course, already tested, more reliable and less error prone. Also it saves time, effort and one of the reusability feature in C++. Because of reusability, the classes distributed as library can be extended to form new class without modifying the library.

6.1 Base and Derived Class



6.2 Protected Access

Access specifiers:

The members are normally grouped into sections private, public protected, called as access spefiers or visibility levels. The members declared in the private sections are hidden from other part of the program and are visible by the member functions within the class. The public access specifiers allows functions or data members in public section to be accessible to other part of the program too. When access specifiers are not specified, members will be private by default.

Following points are to be considered when different access specifiers can be used in inheriting features of base class members.

a. A private member is accessible only to member of the class in which they are declared. They are not visible in derived class or from outside of the class where they are declared except friend functions/ classes.

b. A private member of the base class can be accessed in the derived class through the public member functions of the base class.

c. A protected member is accessible to members of its own and any of the members in a derived class or the friend of base or friend of derived class.

d. The members that might be used in derived class should be declared as protected rather than private.

e. A public member is accessible to members of its own class member of derived class and even outside the class.

Let's first review what we know about the access specifiers private and public. Class members (which can be data or functions) can always be accessed by functions within their own class, whether the members are private or public. But objects of a class defined outside the class can access class members only if the members are public. For instance, suppose an object objA is an instance of class A, and function funcA() is a member function of A. Then in main( ) (or any other function, that is not a member of A) the statement

objA.funcA( );

will not be legal unless func( ) is public. The object objA cannot access private members of class A. **Private** members are, well, private.

This is all we need to know if we don't use inheritance. With inheritance, however, there is a whole raft of additional possibilities. The question that concerns us at the moment is, can member functions of the derived class access members of the base class? The answer is that member functions can access members of the base class of the members are public, or if they are protected. They can't access **private** members.

We don't want to make data member public, since that would allow it to be accessed by any function anywhere in the program, and eliminate the advantages of data hiding. A **protected** member, on the other hand, can be accessed by member functions in its own class or – and here's the key- in any class derived from its own class. It can't be accessed from functions outside these classes, such as main(). This is just what we want.

The moral is that if you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any data or functions that the derived classes might need to access should be made **protected** rather than **private.** This ensures that the class is "inheritance ready".

6.3 Derived Class Declaration

Example: of Ambiguity

```
#include<iostream>
using namespace std;
class A
{
        public:
      void show( ) {cout<<"Class A";}
};
class B
{
        public:
      void show( ) {cout<<"Class B";}
};
class C:public A, public B
{ };
```

```
int main( )
{
    C objC; //object of class C
    //objC.show ();//ambiguous...error
    objC.A::show( );        //Ok invokes show() in class A.
    objC.B::show( );        //Ok
    return 0;
}
```

6.4 Member Function Overriding

If we override a function name in derived class which is in overloaded form in base class, then none of the functions of the base class are accessible after overriding without using the base class name. If we define a function in derive class in the same name of overloaded functions in base class, even with different parameter list, the base class functions are hidden. C++ has a mechanism to access those function. In this case the base member can be accessed with base class name and scope resolution operator before function name.

The process of creating members in the derived class with the same name as that of the visible members of the base class is called overriding. It is called overriding because the new name overrides (hides or displaces) the old name inherited from base class. After overriding, when the members are accessed with overridden names in derived class or through the object of the derive class the derived class members are accessed.

With the overridden function, we can extend the function in derived class which are already define in base class. The same function is used to indicate similar nature of functionality. The derived class's overriding function extends the functionality as according to the derive class. Similarly, data members are overridden for similar functionality in derived class. The data members are rarely overridden than the function members. Even though the overriding hides the base class members, C++ allows mechanism to access the base class members even after overriding.

E.g.,

```
#include<iostream>
using namespace std;
class B     //base class
{  protected:
      int x;
      int y;
   public:
     B( ){ }
     void read( )
     {
    cout<<"X in class B?";
            cin>>x;
            cout<<"Y in class B?";
            cin>>y;
     }
    void show( )
    {
            cout<<"X in class B="<<x<<endl;
            cout<<"Y in class B="<<y<<endl;
    }
};
class D:public B    //publicly derived class
{
   protected:
      int y;
      int z;
   public:
      void read( )
      {
          B::read( );
          cout<<"Yin class D?";
          cin>>y;
          cout<<"Z in class D?";
          cin>>z;
      }
      void show( )
      {
```

```
        B::show( );
        cout<<"Y in class D="<<y<<endl;
        cout<<"Z in class D="<<z<<endl;
        cout<<"Y of B, show from D="<<B::y;//refers to y of
class B
        }
};
int main( )
{
   D objd;
   cout<<"Enter data for object'of class D.."<<endl;
   //objd.base::read( );
   objd.read( );
   cout<<"Contents of object of class D.."<<endl;
   //objd.base::show( );
   objd. show( );
   return 0;
}
```

In the above example with the help of scope resolution operator the base class function can be called in derived class function and with the help of derived class object we can access base class function

How the function over-riding differ from function overloading?

Function over-riding and function overloading are the concept of polymorphism. In function over-riding, the process of creating members in the derived class with the same name as that of the visible members of the base class is called overriding. It is called overriding because the new name overrides (hides or displaces) the old name inherited from base class. After overriding, when the members are accessed with overridden names in derived class or through the object of the derive class the derived class members are accessed.

Whereas in function overloading several functions of the same name to be defined, as long as these functions have different sets of parameters. This capability is called function overloading. An overloaded functions appears to perform different activities depending on the kind of data sent to it. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that performs similar tasks, but on different data types.

6.5 Form of Inheritance: Single, Multiple, Multilevel, Hierarchical, Hybrid and Multipath.

Single Inheritance

When a derived class inherits only from one base class it is known as single inheritance. The general form:

```
class derived_class_name: visibility_mode base_class_name
{ //member of derived class
}
```

Following figure shows the single inheritance pictorially.

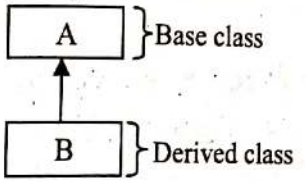


Figure: single Inheritance

```
class A
{...};
class B:public A
{...};
```

Multiple Inheritance

A class can be derived from not only single class but it can be derived more than one class. When a subclass inherits from multiple base classes it is known as multiple inheritance. In multiple inheritances, it combines one or more base class to create a derived class. The syntax of multiple inheritance is :

```
class derived_class_name:vis_mode base1, vis_mode base2, .....
{ //members of derived class
};
```

Following figure show the multiple inheritance:

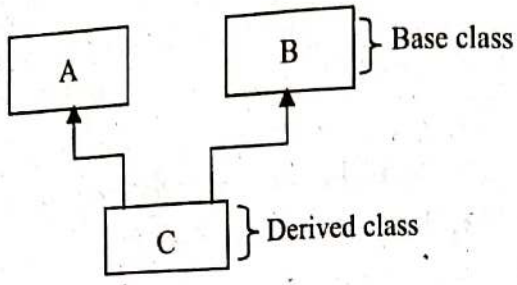


Figure: Multiple Inheritance

```
Class A
{...};
Class B
{...};
Class C:public A, public B
{...};
```

Ambiguity resolution in multiple inheritance:

Ambiguity is a problem that surfaces in certain situation involving multiple inheritances.

- Base classes having function with the same name.
- The class derived from these base classes is not having a function with the name as those of its base classes.
- Member of a derived class or its objects referring to a member, whose name is the same as those in base classes.

The problem of ambiguity is resolved using the scope resolution operator as show in figure.

Objectname.BaseclassName::MemberName(.....)

For Example

```
#include<iostream>
using namespace std;
class student
{
   protected:
        int rno,m1,m2;
   public:
       void get( )
       {
        cout<<"Enter the Roll no :";
          cin>>rno;
          cout<<"Enter the two marks  :";
          cin>>m1>>m2;
          }
};
class sports
{
   protected:
     int sm;              // sm = Sports mark
   public:
     void getsm( )
     {
cout<<"\nEnter the sports mark :";
        cin>>sm;
     }
};
class statement:public student,public sports
{
      int tot,avg;
   public:
     void display( )
     {
       tot=(m1+m2+sm);
       avg=tot/3;
       cout<<"\n\n\t Roll No   : "        <<rno<<"\n\tTotal
        : "<<tot;
        cout<<"\n\tAverage   : "<<avg;
      }
};
int main( )
{
          statement obj;
          obj.get( );
          obj.getsm( );
          obj.display( );
          return 0;
}
```

Multilevel Inheritance

The transitive nature of inheritance id reflected by this form of inheritance. When a derived class inherits from a class that itself inherits from another class, it is known as multilevel inheritance. The general form is:

```
class A
{          .....     };
class B: visibility_mode A
{          ....      };
class C: visibility_mode B
{          ....      };
```

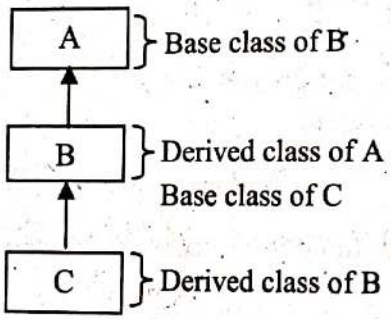


Figure: Multilevel Inheritance

```
class A
{...};
class B:public A
{...};
class C:public B
{...};
```

Hierarchical Inheritance

When many derived classes inherit from a single base class, it is known as hierarchical inheritance. In this form of inheritance the more than one derived class have the common base class. The general form is:

```
class derived1 : visibility_mode base
{          .....     };
class derived2 : visibility_mode base
{          ....      };
class derived3 : visibility_mode base
{          ....      };
```

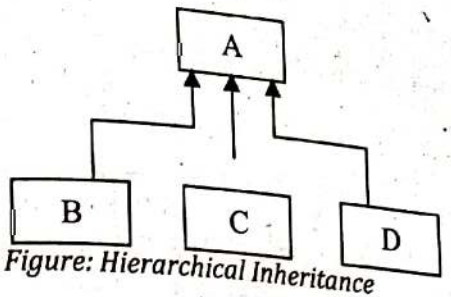


Figure: Hierarchical Inheritance

```
class A
{...};
class B : public A
{...};
class C : public A
{...};
class D : public A
{...};
```

Hybrid Inheritance

When a derived class inherits from multiple base classes and all of its base classes inherit from a single base class, this form of inheritance is known as hybrid inheritance.

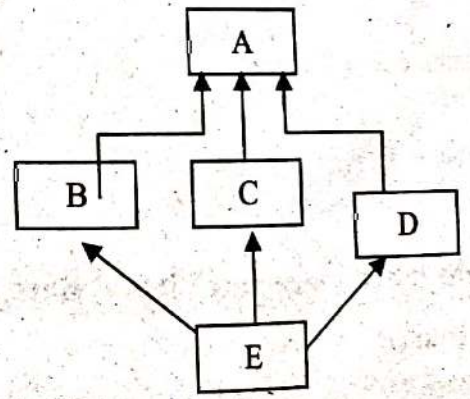


Figure: Hybrid Inheritance

```
class A
{...};
class B:public A
{...};
class C:public A
```

{...};
class E: public B,public C, public D
{...};

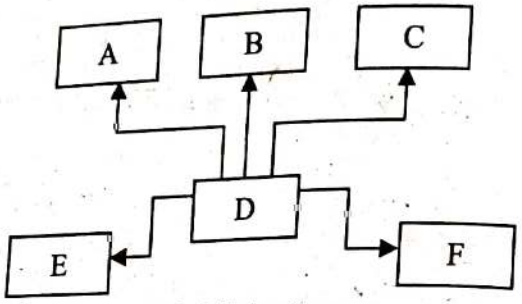


Figure: Hybrid Inheritance

```
class A
{...};
class B
{...};
class C
{...};
class D: public A, public B, public C
{...};
class E; public D
{...};
class F: public D
{...};
```

6.6 Multipath Inheritance and Virtual Base Class

Multipath Inheritance:

When a base class is derived to two or more than two derived class, and these derived is again combine as base class to another derived class, then this type of inheritance is known as Multipath inheritance.

Problem faced in multipath inheritance:

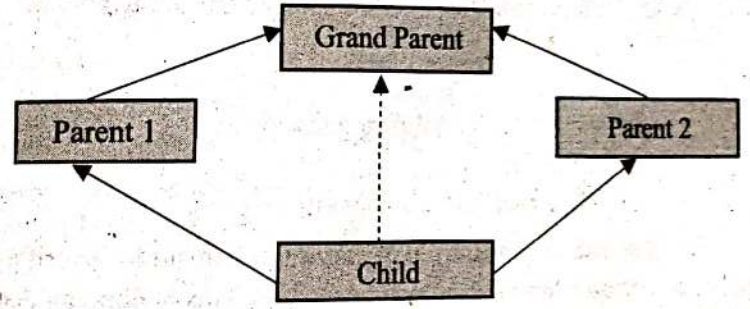
Multipath inheritance can pose some problems in compilation. The public and protected members of grandparent are inherited into the child class twice, first, via parent 1 class and then via parent2 class. Therefore, the child class would have duplicate sets of members of the grand-parent which leads to ambiguity during compilation and it should be avoided.

It can be resolved adding *virtual* to the access specifier

```
class A     //grandparent
{.....};
class B1:virtual public A    //parent1
{.....};
class B2:public virtual A    //parent2
{.....};
class C:public B1,public B2          //child
{.....  //only one copy of A will be inherited
 .....
};
```

Virtual Base Class

Need of Virtual Base class



From this inheritance figure, we see that the **child** class is derived from two base classes **parent 1** and **parent 2** which themselves have a common base class **grandparent**. The **child** inherits the properties of the **grandparent** class via two paths from **parent1** and **parent 2**. For class **child** classes **parent1** and **parent 2** are direct base classes whereas **grandparent** is referred as the indirect base class. The properties of the base class **grandparent** are inherited to **child** class through the two paths so that there are duplicate sets of members of the **grandparent** in the **child**. Because of duplicate sets of members in **child** there will be ambiguity in ascending the **grandparent** members in **child**.

Lets see the following example

```
class grandparent
    {
        private:
                int data;
        public:
                void setdata(int n)
                {data=n;                }
                void showdata( ){cout<<data;}
    };
    class parent1:public grandparent
    { };
    Class parent2:public grandparent
    { };
    class child : public parent1,public parent2
    {
        public:
                void showchdata( )
                    {
            showdata( );//ambiguous which showdata( )
                    }
    };
```

Now consider the following statement

```
child ch;
ch.showdata( );//ambiguity
```

To eliminate this problem C++ has a mechanism to inherit a single copy of properties from the common base class. This is done by declaring the base class as virtual while creating derive classes from this base classes. The error in the above program can be removed by declaring grandparent as virtual base class when creating parent 1 and parent 2 as

```
class grandparent
{
    private:
            int data;
    public:
            void setdata(int n)
            {data=n;                }
            void showdata( ){cout<<data;}
};
class parent1:public virtual grandparent
{ };
class parent2: public virtual grandparent
{ };
class child : public parent1,public parent2
{
    public:
            void showchdata( )
                {
        showdata( );//ambiguous which show data( )
                }
};
```

The keyword **virtual** and public or protected may be used in any order. After adding the keyword virtual while creating classes **parent1** and **parent2** ,it ensures that only one copy of the properties of class **grandparent** is inherited in the class **child** which is derived from classes **parent1** and **parent 2**.

6.7 Constructor Invocation in Single and Multiple Inheritances

Constructor invocation order in single inheritance

```
// Constructor only in the base class
#include<iostream>
using namespace std;
class Base          //base class
{
   public:
     Base( )
      {
       cout<<"No-argument constructor of the base class Base is
                executed";
      }
};
class Derived:public Base          //publicly derived class
{
};
int main( )
{
   Derived obid;          //base constructor
```

```
    return 0;
}
// Constructor only in the derived class
#include<iostream>
using namespace std;
class Base        //base class
{
   //body of base class, without constructors
};
class Derived:public Base//publicly derived class
{
   public:
     Derived( )
     {
        cout<<"Constructors exists in only in derived
              class"<<endl;
     }
};
int main( )
{
    Derived objd;          //access derived constructor
    return 0;
}
// Constructor in both base and derived classes
#include<iostream>
using namespace std;
class Base        //base class
{
   public:
     Base( )
     {
        cout<<"No-argument constructor of the base class Base
              executed first\n";
     }
};
class Derived:public Base//publicly derived class
{
   public:
     Derived( )
     {
        cout<<"No-argument constructors of the derived class D
              executed next"<<endl;
     }
};
int main( )
{
        Derived objd;    //access derived constructor
}
```

Constructor invocation order in multiple inheritance

```
#include<iostream>
using namespace std;
class Base1        //base class
{
   public:
     Base1( )
     {
     cout<<"No-argument constructor of the base class
     Base1";
     }
};
class Base2        //base class
{
   public:
     Base2( )
     {
     cout<<"No-argument constructor of the base class
     Base2";
     }
};
class Derived:public Base2,public Base1//publicly derived class
{
   public:
     Derived()
     {
     cout<<"\nNo-argument constructor of the derived class
           Derived"<<endl;
     }
```

```
}
int main( )
{
    Derived objd;
    return 0;
}
```

6.8 Destructor in Single and Multiple Inheritance.

Destructor in single inheritance

When an object of derived class is created, first the base class constructor is invoked, followed by the derived class constructors. When an object of derived class expires, first the derived class destructor is invoked, followed by the base class destructor.

Constructors and destructors of base class are not inherited by the derived class. Besides, whenever, the derived class needs to invoke base class's constructor or destructor, it can invoke them through explicitly calling them.

Example

```
#include<iostream>
using namespace std;
class base
{
public:
  base( )
  {
    cout<<"\nBase Class Constructor:";
  }
  ~base( )
  {
    cout<<"\nBase Class Destructor:";
  }
};
class Derived : public base
{
public:
  Derived( )
  {
    cout<<"\nDerived Class Constructor;";
  }
  ~Derived( )
  {
    cout<<"\nDerived Class Destructor;";
  }
};
int main( )
{
  {
    Derived D1;
  }
  return 0;
}
```

Output:

```
Base Class Constructor:
Derived Class Constructor;
Derived Class Destructor;
Base Class Destructor:
```

Destructor in Multiple Inheritance

In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. The destructor execute in reverse order to that of constructor i.e. destructor of derived class is called and then the destructor of the base class which is last, in declaration of derived class and followed by base class in reverse order.

Example:

```
#include<iostream>
using namespace std;
class base1
{
public:
  base1( )
  {
```

```
    cout<<"\nBase1 Class Constructor:";
  }
  ~base1()
  {
    cout<<"\nBase1 Class Destructor:";
  }
};
class base2
{
public:
  base2()
  {
    cout<<"\nBase2 Class Constructor:";
  }
  ~base2()
  {
    cout<<"\nBase2 Class Destructor:";
  }
};
class Derived : public base1,public base2
{
public:
  Derived()
  {
    cout<<"\nDerived Class Constructor;";
  }
  ~Derived()
  {
    cout<<"\nDerived Class Destructor;";
  }
};
int main()
{
  {
    Derived D1;
  return 0;
}
```

Output:

```
Base1 Class Constructor:
Base2 Class Constructor:
Derived Class Constructor;
Derived Class Destructor;
Base2 Class Destructor:
Base1 Class Destructor:
```

SOME EXAMPLES

1. Write a program to create classes to represent student, teaching staffs and non-teaching staffs from the base class person. Use proper members in the classes to make your program meaningful. [2070 Ashad]

Answer:

```
#include<iostream>
using namespace std;
const int size=25;
class person
{
      char name[size];
      long int contact_no;
      char address[size];
  public:
      void read()
      {
       cout<<"\nName:";cin>>name;
       cout<<"\nContact Number:";cin>>contact_no;
       cout<<"\nAddress:";cin>>address;
      }
      void write()
      {
       cout<<"\nName:"<<name;
       cout<<"\nNumber:"<<contact_no;
       cout<<"\nAddress:"<<address;
```

```
};        //end person
class student:public person
{
   long int roll_no;
  public:
   void read( )
   {
    person::read( );
    cout<<endl<<"Enter Student Details:"<<endl;
    cout<<"Rollno:\n";cin>>roll_no;
    }
   void write( )
   {
    cout<<"\nStudent Information:"<<endl;
    person::write( );
    cout<<"\nRollno:"<<roll_no;
   }
};
class teachingstaffs:public person
{
    char depart[size];
    char design[size];
  public:
    void read( )
    {
     person::read( );
     cout<<"Enter Department";
     cin>>depart;
     cout<<"Enter Designation:";
     cin>>design;
     }
    void write( )
     {
      person::write( );
      cout<<"\nDepartment:"<<depart<<endl;
      cout<<"\nDesignation:"<<design;
     }
};
class nonteachingstaffs:public person
{
   char post[size];
  public:
   void read( )
   {
    person::read( );
    cout<<"\nEnter Post:"<<endl;
    cin>>post;
   }
   void write( )
   {
    person::write( );
    cout<<"\nPost:"<<endl<<post<<endl;
   }
};
int main( )
{
   student s1;
   s1.read( );
   s1.write( );
   teachingstaffs t1;
   cout<<"Enter Teaching Staffs Details:\n";
   t1.read( );
   cout<<"Teaching Staffs Information:"<<endl;
   t1.write( );
   nonteachingstaffs n1;
   cout<<"Enter Non-Teaching Staffs Details:\n";
   n1.read( );
   cout<<"Non-Teaching Staffs Information:"<<endl;
   n1.write( );
   return 0;
}
```

2. Write a program with two base class and a third class inherited from the two base classes. Assume that both of the two base classes have two data members name and age. Add other suitable functions so that base classes are student and employee and the derived class is manager. **[2066 Jestha]**

Answer:

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

```
class student
{
  protected:
  char name[10];
        double aggr;
        int age;
  public:
  void display( )
    {
  cout<<"\nName:"<<name;
     cout<<"\nAge:"<<age;
     cout<<"\nAggregiate:"<<aggr;
     }
};
class employee
{
  protected:
       char name[10];
       int age;
       double salary;
     public:
    void display( )
       {
         cout<<"\nName:"<<name;
         cout<<"\nAge:"<<age;
         cout<<"\nSalary:"<<salary;
       }
};
class manager:public student,public employee
{
  char company[20];
  public:
```



```
void input( )
 {
   cout<<"\nEnter Name:";
   cin>>student::name;
   cout<<"\nEnter Age:";
   cin>>student::age;
   cout<<"\nEnter Aggregiate:";
   cin>>student::aggr;
   cout<<"\nEnter salary:";
   cin>>salary;
   cout<<"\nEnter Company Name:";
   cin>>company;
 }
 void display( )
 {
   cout<<"\nSalary:"<<salary;
   cout<<"\nCompany Name:"<<company;
 }
};
int main( )
{
  manager m;
        m.input( );
        cout<<"\nThe manager Record:";
    m.student::display( );
    m.display( );
    return 0;
}
```

3. Write a program with a class cricketer that has data members to represent name, age, and no of matched played. From this class cricketer derive two classes, bowler and batsman. The bowler class should have no of wickets as data members and the batsman class should have no of runs and no of centuries as data members. Use appropriate member functions in all classes to make the program meaningful. [2067 Ashad]

Answer:

```
#include<iostream>
using namespace std;
class cricketer
{
  protected:
    char name[40];
    int age;
    int no_of_match;
  public:
    void getinfo( )
    {
 cout<<"Your Name:";
        cin>>name;
        cout<<"\nYour Age:";
        cin>>age;
        cout<<"\nEnter total number of match played:";
        cin>>no_of_match;
    }
    void showinfo( )
    {
 cout<<"\nName:"<<name;
        cout<<"\nAge:"<<age;
        cout<<"\nMatched played:"<<no_of_match;
    }
};
class bowler:public cricketer
{
  protected:
    int no_of_wickets;
  public:
    void getinfo( )
```



```
    {
 cricketer::getinfo( );
        cout<<"How many wickets taken till now:";
        cin>>no_of_wickets;
    }
    void showinfo( )
    {
 cricketer::showinfo( );
        cout<<"\nNumber of wickets:"<<no_of_wickets;
    }
};
class batsman:public cricketer
{
  protected:
 int no_of_runs;
        int no_of_centuries;
  public:
 void getinfo( )
        {
 cricketer::getinfo( );
        cout<<"\nHow many number of runs till now:";
        cin>>no_of_runs;
        cout<<"\nHow many number of centuries:";
        cin>>no_of_centuries;
        }
        void showinfo( )
        {
 cricketer::showinfo( );
        cout<<"\nNumber of runs:"<<no_of_runs;
        cout<<"\nNumber of centuries:"<<no_of_centuries;
        }
};
```

```
int main( )
{
  bowler bow1;
  batsman bat1;
  cout<<"\nEnter Bowler information:"<<endl;
  bow1.getinfo( );
  cout<<"\nDetails of Bowler:"<<endl;
  bow1.showinfo( );
  cout<<"\nEnter Batsman information:";
  bat1.getinfo( );
  cout<<"\nDetals of Batsman:"<<endl;
  bat1.showinfo( );
  return 0;
}
```

4. Write a program with three classes students, test and result by using multilevel inheritance. Assume necessary data members and functions yourself and program with input information, input data and calculate marks total and display result. [2070 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class students
{
  private:
    char *name;
    int roll_no;
  public:
    void input_data( )
    {
      cout<<"What is your name?"<<endl;
      cin>>name;
      cout<<"Your College Roll Number:"<<endl;
      cin>>roll_no;
    }
    void display( )
    {
      cout<<"Name:"<<name<<endl;
      cout<<"Roll:"<<roll_no;
    }
};
class test:public students
{
  protected:
    int oop,logic,EDC;
  public:
    void input_data( )
    {
      students::input_data( );
      cout<<"Enter marks of:"<<endl;
      cout<<"OOP:"<<endl;
      cin>>oop;
      cout<<"Digital Logic:"<<endl;
      cin>>logic;
      cout<<"Electronic Devices and Circuit:"<<endl;
      cin>>EDC;
    }
    void display( )
    {
      cout<<endl;
      students::display( );
      cout<<endl<<endl;
      cout<<"Marks in:"<<endl;

      cout<<"**********************************"<<end
l;
      cout<<"Object Oriented Programming:"<<oop;
      cout<<endl<<"Digital Logic"<<logic;
cout<<endl<<"Electronic Devices and Circuit"<<EDC;
    }
};
class result:public test
{
  protected:
    float percent;
```

```
public:
  void input_data( )
  {
     test::input_data( );
  }
  void calculate( )
  {
     percent=((oop+logic+EDC)*100)/3;
  }
  void display( )
  {
     test::display( );
     cout<<endl<<"Percentage:"<<percent<<"%";
  }
};
int main( )
{
  result r;
  r.input_data();
  r.calculate();
  r.display();
  return 0;
}
```

5) Write a program to show the exception order of constructor and destructor in multilevel inheritance. Show your program output. **[2073 Shrawan]**

Answer:

```
#include<iostream>
using namespace std;
class Book
{
protected:
  int number;
  double price;
public:
  Book(int a, double b)
  {
     number=a;
     price=b;
  }
  ~Book( )
  {
     cout<<"\nBase class Destructor\n";
  }
};
class SubClass:public Book
{
protected:
  int pages;
public:
  SubClass(int p,double q, int r):Book(p,q)
  {
     pages=r;
  }
  ~SubClass( )
  {
     cout<<"\nSubClass Destructor\n";
  }
};
class SubSubClass:public SubClass
{
protected:
  double discount;
public:
  SubSubClass(int c, double d, int e, double f):SubClass(c,d,e)
  {
     discount=f;
  }
  ~SubSubClass( )
  {
     cout<<"\nSubSubClass Destructor\n";
  }
  void display( )
```

```
    {
      cout<<"\n Number="<<number;
      cout<<"\nPrice="<<price;
      cout<<"\nPages="<<pages;
      cout<<"\nDiscount="<<discount;
    }
};
int main( )
{
   SubSubClass objD(1,240.0,405,0.3);
   objD.display( );
   return 0;
}
```

6. Write a program to show the order of constructor invocation in multilevel inheritance. [2074 Ashwin]

Answer:

```
#include<iostream>
using namespace std;
class base
{
public:
   base( )
   {
      cout<<"Constructor in base class"<<endl;
   }
   ~base( )
   {
      cout<<"Destructor in base class"<<endl;
   }
};
class derived:public base
{
public:
   derived( )
   {
      cout<<"Constructor in derived class"<<endl;
   }
   ~derived( )
   {
      cout<<"Destructor in derived class"<<endl;
   }
};
class dderived:public derived
{
public:
   dderived( )
   {
      cout<<"Constructor in dderived class"<<endl;
   }
   ~dderived( )
   {
      cout<<"Destructor in dderived class"<<endl;
   }
};
int main( )
{
   dderived d;
   return 0;
}
```

Output:

```
Constructor in base class
Constructor in derived class
Constructor in dderived class
Destructor in dderived class
Destructor in derived class
Destructor in base class
```

CHAPTER 7:

POLYMORPHISM AND DYNAMIC BINDING

Polymorphism basically means many forms, so it stands to reason that any class which has many forms is polymorphic. Therefore, a template class is polymorphic, a class with virtual functions is polymorphic, even a class with overloaded member functions is polymorphic.

Efficiency vs. flexibility are the primary tradeoffs between static & dynamic binding Static binding is generally more efficient since

1. It has less time & space overhead
2. It also enables method in lining

Dynamic binding is more flexible since it enables developers to extend the behavior of a system transparently

However, dynamically bound objects are difficult to store in shared memory

Compile- Time Binding

In compile time, the compiler at the compile time know all the matching arguments, therefore the compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking. Also know as compile time polymorphism.Early binding simply means that an object is bound to its function call at compile time.

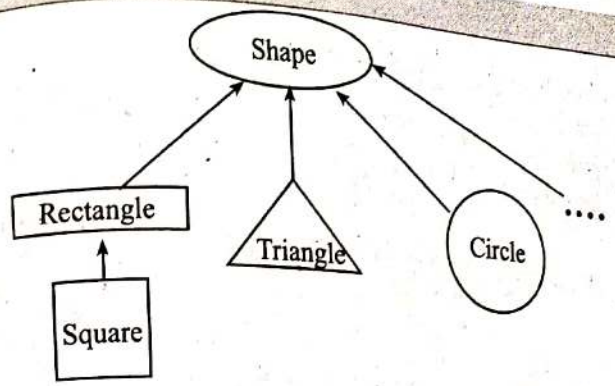
Runtime Binding

If appropriate member function are chosen at run time rather than compile time, this is known as runtime polymorphism or late binding or runtime binding or dynamic binding. Dynamic binding (also known as runtime 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

Concrete Class

The base class which has at least one pure virtual function is called abstract base class or just abstract class. Contrary to these, the classes whose objects can be created are called concrete class.

7.1 Need of Virtual Function



Need of Virtual function:

Suppose you have a number of objects of different classes but you want to put them all on a list and perform a particular operation on then using the same function call. Then it is better to use virtual function.

For example:

```
#include <iostream>
using namespace std;
class CPolygon
{
        protected:
        int width, height;
        public:
                void set_values (int a, int b)
          {
width=a; height=b;
}
                virtual int area ( )
          {
return (0);
}
 };
class CRectangle: public CPolygon
{
  public:
  int area ( )
    {
return (width * height);
```

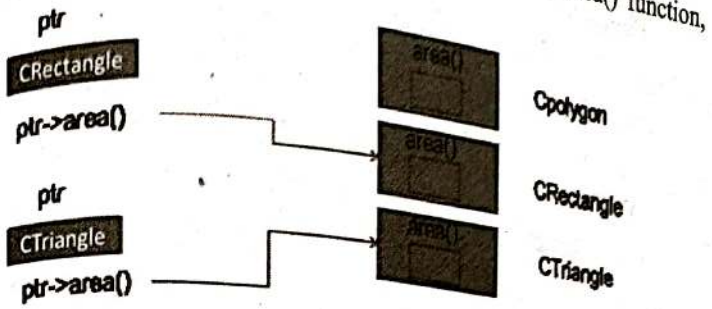
```
}
};
class CTriangle: public CPolygon
{
  public:
  int area ( )
   {
return (width * height / 2);
}
};
int main ( )
{
  CRectangle rect;  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly;
  ppoly = &rect;
  ppoly->set_values (4,5);
  cout << ppoly->area( ) << endl;
  ppoly = &trgl;
  ppoly->set_values (4,5);
  cout << ppoly->area( ) << endl;
  ppoly = &poly;
  ppoly->set_values (4,5);
  cout << ppoly->area( ) << endl;
  getch( );
}
```

Explanation:

What the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example. So the same function call

```
ppoly->area( );
```

executes different functions, depending on the contents of ptr. The rule is that the compiler selects the function according to the contents of the pointer ptr, not on the type of the pointer.A class that declares or inherits a virtual function is called a *polymorphic class*.Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to call its own area() function, which always returns 0.



7.2 Pointer to Derived Class

Whenever we create an object of a class, another pointer called virtual pointer or vptr is created. Each object created has its unique vptr associated with it, which points to the vtbl of the corresponding class. The vptr is also not visible to the programmer. From the vtbl the run-time system can find out the address fof the function of the class. Thus, a link between the specific object and the address of the function is clearly established.

When we implement the virual functions, we first assign one of the derived class objects to point to the base class pointer. When this is carried out, the vptr of the derived object will in turn be linked to the base class pointer. Next, when we call the function with the base class pointer, the exact function can be linked through the respective vptr.

The pointer to object of base class or the derived class can be created. As derived class are the type of the base class the derived class pointer (address) are also type of base class pointer. That is base class pointers are type compatible with derived class pointer, allowing derived class pointer to be used as base class pointer. So a base class pointer can hold the address of the derived class but the reverse is not true. Because of the compatibility of base class pointer to derived class pointer, the single base class pointers can be used to hold the address of base class objects or the derived class object. The runtime polymorphism or dynamic binding in C++ is possible because the base class pointer can hold the address of its own class as well as the address of its derived class.

Example:

```
Animal *panm;
Animal anm;
Cow cw;
Dog dg;
panm=&anm;
panm->display( );
panm=&cw;
panm->display( );
panm=&dg;
panm->display( );
```

7.3 Definition of Virtual Functions

Virtual function:

When we use the function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtualpreceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

7.4 Array of Pointers to Base Class

The array of pointers to base class objects is used to store pointer to objects of different derived classes of that base class. The common interface function in all the classes is declared as virtual in the base class and defined as a normal function in all other derived classes. With virtual function in base class when accessing same member function through elements of base class pointer pointing to different objects, different function related to those objects are called. So the base class pointer responds to same message (function call) differently.

Example:

```
#include <iostream.h>
using namespace std;
class CPolygon
{
protected:
  int width, height;
public:
  void set_values (int a, int b)
    { width=a; height=b; }
  virtual int area ( )
    { return 0; }
};
class CRectangle: public CPolygon
{
public:
  int area ( )
    { return (width * height); }
};
class CTriangle: public CPolygon
{
public:
  int area ( )
    { return (width * height / 2); }
};
int main ( )
{
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  Cpolygon *cpoly[ ]={&poly,&trgl,&rect};
  for(int j=0;j<3;j++)
  cpoly[j]->set_values (4,5);
  cout<<"Figure drawn by base pointer are:"<<endl;
  for(int i=0;i<3;i++)
  cout << cpoly[i]->area( ) << endl;
  return 0;
}
```

7.5 Pure Virtual functions and Abstract Class

Pure virtual function:

A pure virtual function is a virtual function with no body. Every concrete derived class must override all base-class pure virtual functions and provide concrete implementations of those functions. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving runtime polymorphism.

A pure virtual function is one with an initialize of = 0 in its declaration, for example

```
virtual void draw( ) const = 0;
```

The difference between a virtual function and pure virtual function is that a virtual function has an implementation and gives the derived class the option of overriding the function; by contract, a pure virtual function does not provide an implementation and requires the derived class to override the function.

Abstract class:

Pure virtual function is called abstract class. The main objective of an abstract base class is to provide some trails to the derived classes and to create a base pointer required for achieving run time polymorphism.

Normally, when creating class hierarchy with virtual functions, in most of the cases it seems that the base class pointers are used but the base class objects are rarely created. When the objects of base class are never instantiated, such a class is called abstract base class or simply abstract class. Such a class only exists to act as a parent of derived classes from which objects are instantiated. It may also provide interface for class hierarchy. Abstract classes can be used as a framework upon which new classes can be built to provide new functionality. Also, it can be used to set reusable classes so that programmer can extend them.

7.6 Virtual Destructor

Need of virtual destructor:

Base class destructors should always be virtual. Suppose you use delete with a base class pointer to a derived class object to destroy the derived-class object. If the base-class destructor is not virtual, then delete, like a normal member function, calls the destructor for the base class, not the destructor for the derived class. This will cause only the base part of the object to be destroyed.

If none of the destructors has anything important to do (like deleting memory obtained with new) then virtual destructors aren't important. But in general, to ensure the derived-class objects are destroyed properly, you should make destructors in all base classes virtual.

For example:

```
#include<iostream>
using namespace std;
class Base
{
public:
 ~Base( )                    //non-virtual destructor
 // virtual ~Base( )         //virtual destructor
  {
     cout<<"Base destroyed\n";
  }
};
class Derived:public Base
{
  public:
   ~Derived( )
   {
     cout<<"Derived destroyed\n";
   }
};
int main( )
{
   Base* pBase=new Derived;
   delete pBase;
   return 0;
}
```

Output will be: Base destroyed

This shows that the destructor for the **Derived** part of the object isn't called. In the listing the base class destructor is not virtual, but you

can make it so by commenting out the first definition for the destructor and substituting the second.

```
#include<iostream>
using namespace std;
class Base
{
  public:
     //~Base( )            //non-virtual destructor
     virtual ~Base( )      //virtual destructor
        {
        cout<<"Base destroyed\n";
        }
};
class Derived:public Base
{
    public:
      ~Derived( )
        {
        cout<<"Derived destroyed\n";
        }
};
int main( )
{
   Base* pBase=new Derived;
   delete pBase;
   return 0;
}
```

Output will be: Derived destroyed
Base destroyed

7.7 Reinterpret_cast Operator

The reinterpret_cast operator is used to convert one type into a fundamentally different type. For example it can be use to change pointer type object to integer type object and vice versa. It can be used for casting inherently incompatible pointer types. The general form is:

```
reinterpret_cast<type>(object);
```

Here, type specifies the target type of the cast and the object being cast into the new type. The reinterpret cast converts the object from one type to another type without checking the bit pattern or size of source and destination type. If the number of bit of source and destination is same, then the conversion in vice versa takes place properly but if there is difference in bit than the conversion in vice versa doesn't take properly.

7.8 Run-Time Type Information

In C++, it is possible to find out information about an object's type and even change the type of the object at runtime. Because of the ability of finding the object's type and changing the type of the object at runtime, this feature is called run-time type information.

The operator dynamic_cast and typeid provides us the run_time type information.

Usually, the runtime type information is used in situation where a variety of classes are derived from the base class. For dynamic_cast operator to work, the base class must be polymorphic, that is, the base class must have virtual function. The dynamic_cast operator is used to convert types between objects of derived class and base class. The typeid operator is useful for polymorphic classes; however it can be used for determining type for any data at runtime.

7.8.1 dynamic_cast Operator

Program code: dynamic_cast operator

dynamic_cast(example)

```
#include<iostream>
using namespace std;
class base
{ public:
      virtual void fun1( )  { }
};
class myclass:public base
{ };
class yourclass:public base
{ };
int main( )
{
  base *b;
    Myclass m, *mp;
    B=&m;
```

```
    if(mp=dynamic_cast<myclass*>(b))
cout<<endl<<"Of type myclass";
    else
cout<<endl<<"Not of type myclass";
    yourclass y;
    b=&y;
    if(mp=dynamic_cast<myclass*>(b))
cout<<endl<<"Of type myclass";
    else
cout<<endl<<"Not of type myclass";
return 0;
}
```

7.8.2 typeid Operator

Program code: type_id operator

```
typeid(example)
#include<typeinfo.h>
#include<iostream.h>
using namespace std;
class base
{
   public:
        virtual void fun1( )        { }
};
class myclass:public base
{ };
class yourclass:public base
{ };
int main()
{
        base *b1;
        cout<<endl<<typeid(b1).name();
        myclass m;
        b1=&m;
        cout<<endl<<typeid(*b1).name();
        base *b2;
        yourclass y;
        b2=&y;
        cout<<endl<<typeid(*b2).name();
        if(typeid(*b1)==typeid(*b2))
        cout<<endl<<"Equal";
        else
        cout<<endl<<"Unequal";
        cout<<endl<<typeid(45).name( );
        cout<<endl<<typeid('4').name( );
        cout<<endl<<typeid(4.5).name( );
return 0;
}
```

SOME EXAMPLES

1. Write a program having student as an abstract class and create derived class such as Engineering, Science and Medical. Show the use of virtual functions in this program. [2070 Ashad]

Answer:

```
#include<iostream>
using namespace std;
class Student
{
   public:
        virtual void show( )=0;//pure virtual function
};
class Engineering:public Student
{
   public:
        void show( )
        {
        cout<<"\n I am Engineering class";
        }
};
class Science:public Student
{
   public:
        void show( )
        {
         cout<<"\n I am Science class";
        }
```

```
};
class Medical:public Student
{
   public:
        void show( )
        {
        cout<<"\nI am Medical Class";
        }
};
int main( )
{
        Engineering e1;
        Science s2;
        Medical m1;
        Student *sptr;    //pointer to base class
        sptr=&e1;         //address of Engineering in pointer
        sptr->show( );    //called from Engineering
        sptr=&s2;
        sptr->show( );  //called from Science
         sptr=&m1;
        sptr->show( );  //called from Medical
   return 0;
}
```

2. Create a derived class manager from two base classes' person and employee. Assume suitable data members in each class and display the information.

Answer:

```
#include<iostream>
using namespace std;
class person
{
   char name[40];
   int age;
  public:
   void getinfo( )
    {
        cout<<"Enter Name:"<<endl;
        cin>>name;
        cout<<"Enter Age:"<<endl;
        cin>>age;
    }
  void display( )
   {
        cout<<"Name:"<<name<<endl;
        cout<<"Age:"<<age<<endl;
    }
};
class employee
{
    int emp_id;
    char position[40];
    int salary;
  public:
    void getinfo( )
     {
        cout<<"Enter Employee Id."<<endl;
        cin>>emp_id;
        cout<<"Enter Position:"<<endl;
        cin>>position;
        cout<<"Enter Salary ammount:"<<endl;
        cin>>salary;
      }
      void display( )
       {
        cout<<"Employee Id is:"<<emp_id<<endl;
        cout<<"Position:"<<position<<endl;
        cout<<"Salary:"<<salary<<endl;
       }
};
class manager:public person,public employee
{
  char m_name[40];
public:
  void getinfo( )
   {
    cout<<"Enter name of transaction manager:"<<endl;
```

```
    cin>>m_name;
  }
  void display( )
  {
    cout<<"Transaction manager name:"<<m_name;
  }
};
int main( )
{
  manager m;
  m.person::getinfo();
  m.employee::getinfo();
  m.getinfo();
  m.person::display();
  m.employee::display();
  m.display();
  return 0;
}
```

3) Write a program having Polygon as an abstract class with Length and Height as its data member. Create derived class Rectangle and Triangle. Make Area() as pure virtual function and redefine it in derived class to calculate respective area. [2072 Chaitra]

Answer:

```
#include<iostream>
using namespace std;
class Polygon
{
 protected:
   float length;
   float height;
 public:
   Polygon( ){length=0;height=0;}
   Polygon(float l, float h)
   {
     length=l;
     height=h;
   }
   virtual float area()=0;
};
class Rectangle:public Polygon
{
 public:
   Rectangle( ){ }
   Rectangle(float l, float h)
   {
     length=l;
     height=h;
   }
   float area( )
   {
     return length*height;
   }
};
class Triangle:public Polygon
{
 public:
   Triangle( ){ }
   Triangle(float l,float h)
   {
     length=l;
     height=h;
   }
   float area( )
   {
     return(0.5*length*height);
   }
};
int main( )
{
  Polygon *plg;
  Rectangle rec(2.5,7.9);
  plg=&rec;
  cout<<"Area of rectangle is:"<<plg->area( );
  Triangle trg(5,6.5);
  plg=&trg;
  cout<<"Area of triangle is:"<<plg->area( );
  return 0;
}
```

CHAPTER 8:

STREAM COMPUTATION FOR CONSOLE AND FILE INPUT/ OUTPUT

8.1 Stream Class Hierarchy for Console Input/ Output

A stream acts as an interface between the program and the input/output device cin and cout are predefined streams which represents the input stream connected to the standard input device(usually the keyboard) and the output stream connected to the standard output device(usually the screen) respectively.

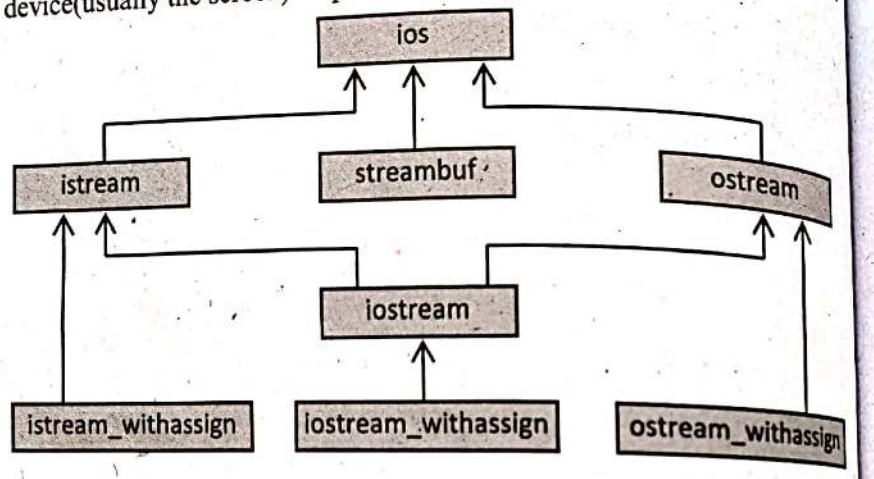


Fig: Stream classes for console I/O operation

File stream:

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream. In other words, the input stream extracts (or reads) data from the file and output stream inserts (or writes) data to the file.

8.2 Testing Stream Errors

The object cin and cout are normally used for input and output. An error can occur while input or displaying data so these errors can be handle by checking the state of stream state. Following member function of ios class can be use to test the stream state:

Function	Descriptions
eof()	Returns true (non-zero value) if end of file is encountered. Otherwise returns false (zero).
fail()	Return true when an input or output has failed.
bad()	Return true if an invalid operation is attempted or any unrecoverable error has occurred.
good()	Returns true if no error has occurred. If file.good() is true, all is well with the stream file and we can procee to perform I/O operations. When it returns false, no further operations can be carried out.

Example:

```
ifstream infile;
infile.open("ABC");
while(!infile.fail( ))
{
  //process the file
}
if(infile.eof( ))
{
  //terminate program normally
}
else if(infile.bad( ))
{
  //report fata error
}
else
{ infile.clear( ); //clear error state so further opearations can be
attemped
}
```

8.3 Unformatted Input/ Output

The four formatted input/output ios member function are:

width( ):

This function of ios class is used to define the width of the field to be used while displaying the output. It is normally accessed with cout object. It has the following two forms:

```
cout.width( );        //gets field width
```

```
cout.width(6);      //sets field width
```

E.g.,

```
cout.width(6);
cout<<849;
output: _ _ _ 8 9 4
```

fill( ):

This ios function is used to specify the character to be displayed in the unused portion of the display width. By default, blank character is displayed in the unused portion. It has two forms:

```
cout.fill( );
cout.fill(character);  //here character can be any special or alphabet
                       characters
```

E.g.,

```
int x=456;
cout.width(6);
cout.fill("#");
cout<<x<<endl;
output: # # # 4 5 6
```

precision( ):

This function belonging to ios class is used to specify maximum number of digits to be displayed as a whole in floating point number or the maximum number of digits to be displayed in the fractional part of the floating point number. In general format precision() specifies the maximum number of digits including fractional or integer parts. This is because in general format the system chooses either exponential (scientific) or normal (fixed) floating point format which best preserves the value in the space available.

```
cout.precision();   //returns the current floating point precision.
cout.precision(4);  //sets upto 4 decimal places after dot.
```

E.g.,

```
float x=5.5005,y=66.769;
cout.precision(3);
cout<<x<<endl;
cout<<y<<endl;
output:
5.5
66.8
```

setf( ):

C++ provides other ways of output formatting like left justified, scientific form, show positive sign, show base of displayed number etc. The ios member function setf() is used to set flags and bit-fields that controls the output in other ways.

```
cout.setf(flag_value, bit_field_value);
```

E.g.,

```
int x=456;
cout.setf(ios::left, ios::adjustfield);
cout.width(6);
cout.fill("#");
cout<<x<<endl;
output: 4 5 6 # # #
```

8.4 Formatted Input/ Output with ios Member Function and Flags

It is seen that for formatted input/output the stream class ios function are used. To call ios function for formatting, we need to write separate statement and call the function through stream objects cin and cout. Manipulators are the formatting function for input/ output that are embedded directly to C++ input/ output statements so that extra statements are not required. Manipulator are used along with the extraction >> and insertion << operators for stream input and output formatting. They can be put together along with input/ output stream to modify the forms of parameters of a stream.

Formatting functions and flags in ios class

Class name	Contents
ios(General input/output stream class)	• Contains basic facilities that are used by all other input and output classes
istream(input stream)	• Inherits the properties of ios.

Class name	Contents
	• Declares input functions such as get( ), getline( ) and read( ). • Contains overloaded extraction operator (>>)
ostream(output stream)	• Inherits the properties of ios. • Declares output functions put( ) and write( ). • Contains overloaded insertion operator(<<).
iostream(input/output stream)	• Inherits the properties of ios, istream and ostream and thus contains all the input and output functions and operators
streambuf	• Holds the characters written or read from input devices before they are sent to actual destination.

8.5 Formatting with Manipulators

Manipulators:

The header file 'iomanip' provides a set of functions called 'manipulators' which can be used to manipulate the output formats.They provide the same features as that of the ios member functions and flags.

Standard manipulators:

- The **header file 'iomanip'** provides a set of functions called 'manipulators' which can be **used to manipulate the output formats.**
- They **provide the same features as that of the ios member functions and flags**
- We can use two or more manipulators as a chain in one statement

cout<<manip1<<manip2<<item;

cout<<manip1<<manip2<<item<<manip3;

Non-parameterized manipulators

It do not take argument to control the formatting of input/output.

Manipulators	Effect produced
endl	output new line and flush
left	sets ios::left flag of ios::adjustfield
right	sets ios::right of ios::adjustfield
dec	sets ios::dec of ios::adjustfield
Hex	sets ios::hex of ios::adjustfield
Oct	sets ios::oct of ios::adjustfield
showpoint	sets ios::showpoint flag
scientific	sets ios::scientific flag of ios::floatfield
fixed	sets ios::scientific of ios::floatfield

Parameterized manipulators

It takes argument for formatting

Manipulators	Effect produced
setw(int n)	Equivalent to ios function width( )
setprecision(int n)	Equivalent to ios function precision( )
setfill(char c)	Equivalent to ios function fill( )
setiosflags(flag)	Equivalent to ios function setf( )
resetiosflags(flag)	Equivalent to ios function unsetf( )

8.6 Stream Operator Overloading

Example:

```
#include<iostream>
using namespace std;
class complex
{
        int real;
        intimag;
  public:
        friend istream& operator>>(istream&, complex &);
        friend ostream& operator<<(ostream&, complex &);
};
istream& operator>>(istream&input,complex&c)
{
    cout<<"\nEnter the real part:";
      input>>c.real;
```

```
        cout<<"\nEnter the imaginary part:";
        input>>c.imag;
      return input;
}
ostream& operator<<(ostream&output,complex&c)
{
    output<<endl<<c.real<<"+i"<<c.imag;
    return output;
}
int main( )
{
      complex c1;
      cin>>c1;
      cout<<"\nThe complex number is :";
      cout<<c1;
   return 0;
}
```

8.7 File Input/ Output with Streams

All of the program presented so far, take input from standard input (normally keyboard) and output displayed on standard output (normally monitor). The console stream object like cout and cin have been used for output and input respectively. However , many applications may require a large amount of data to be read, processed, and also saved for later use. Such information is stored on the auxiliary memory device in the form of data file. And a file is a collection of bytes that is given a name. In most computer systems, files are used as a unit of storage primarily on floppy-disk or fixed disk data storage system (or they can be CDs or other storage devices. Thus data files allow us to store information permanently, and to access and alter that information whenever necessary.

The fstream library predefines a set of operations for handling file related input and output. It defines certain clases that help obe perform file input and output. For example, ifstream class ties a file to the program for input; ofstream class ties a file to the program for output; and fstream class ties a file to the program for both input and output.

8.8 File Stream Class Hierarchy

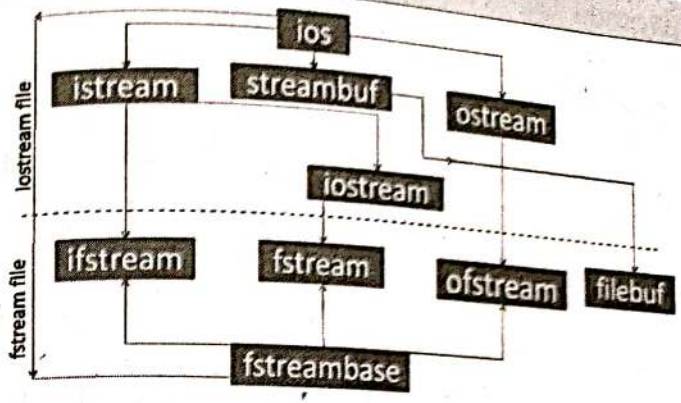


Fig.: Stream classes for file operations

8.9 Opening and Closing Files

A file can be opened in two ways:

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

- The first method is useful when we use only one file in the stream.
- The second method is used when we want to manage multiple files using on stream.

Opening Files Using Constructor:

- We know that a constructor is used to initialize an object while it is being created.
- The creation and assignment if file name to the file stream object involves the following steps:
  - Create a file stream object using the appropriate class depending on the type of file stream required.
    eg, ifstream can be used to create the input stream, *ofstream* can be used to create the output stream, and *fstream* can be used to create the input and output stream.
  - Bind the file stream to the disk. In disk, file stream is identified by a file name.
    File-stream-class stream-object ("filename");
    ifstream infile("database");

- It creates *infile* as the object of the class *ifstream* that manages the input stream, and opens the file *database* and binds it to the output stream disk file.

  ofstream outfile("data.out");

- It defines *outfile* as the object of the class *ostream*, and binds it to the file *data.out* for wirting.

For instance, to print the message *Hello World* on the console and into the file, the following commands can be issued:

cout<<"Hello World";

Prints the message *Hello World* on the standard output devices. Where as the statement

myfile<<"Hello World";

Prints the message *Hello World* into the file pointed to by the file pointer myfile

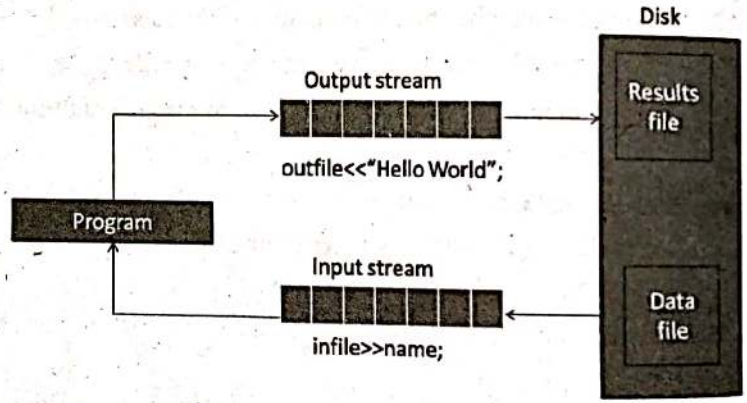


Fig.: File I/O with stream operators

Opening Files Using open( ):

The file can also be opened explicitly using the function open() instead of a constructor.

This mechanism is generally used when different files are to be associated with the same object at different times

This is done as follows:

file-stream-class stream-object;

stream-object.**open**("filename");

Example:

```
ofstream outfile;          //Create stream (for output)
outfile.open("DATA1");     //Connect stream to DATA1
.....
outfile.close( );          //Disconnect stream from DATA1
outfile.open("DATA2");     //Connect stream to DATA2
.....
outfile.close( );          //Disconnect stream from DATA2
.....
```

8.10 Read/ Write from File

1. Insertion Operator (<<) and Extraction Operator (>>)

ifstream infile("database");

- It creates *infile* as the object of the class *ifstream* that manages the input stream, and opens the file *database* and binds it to the output stream disk file.

  ofstream outfile("data.out");

- It defines *outfile* as the object of the class *ostream*, and binds it to the file *data.out* for wirting.

For instance, to print the message *Hello World* on the console and into the file, the following commands can be issued:

cout<<"Hello World";

Prints the message *Hello World* on the standard output devices. Where as the statement

myfile<<"Hello World";

Prints the message *Hello World* into the file pointed to by the file pointer myfile.

2. Sequential Input and Output operations:

put( ):

The function put( ) writes a single character to the associated stream.

Example:

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

```
#include<string>
using namespace std;
int main( )
{
   char str[ ]="This is the test program, for put() function";
   ofstream outfile("Test.txt");      //create file for output
   for(int j=0;j<strlen(str);j++)     //for each character
   outfile.put(str[j]);               //write it to file
   return 0;
 }
```

get():

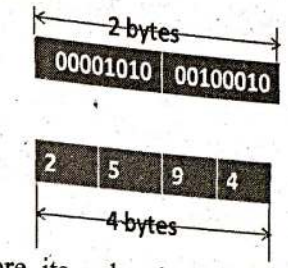
The function get( ) reads a single character from the associated stream.

Example:

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
int main( )
{
 char ch;
 ifstream infile("Test.txt");
 while(infile)
 {
        infile.get(ch);
        cout<<ch;
 }
        return 0;
}
```

3. write( ) and read( ) functions:

The functions write( ) and read( ), unlike the functions put( ) and get( ), handle the data in binary form.



An **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form.

The binary input and output functions takes the following form:

infile.**read**((char *) & v, sizeof(v));

outfile.**write**((char*) & v,sizeof(v));

- These function take two arguments.
- The first is the address of the variable V,and
- The second is the length of that variable in bytes.
- The address of the variable must be cast to type char*(i.e. pointer to character type).

8.11 File Access Pointers and their Manipulators

File pointers and their manipulators

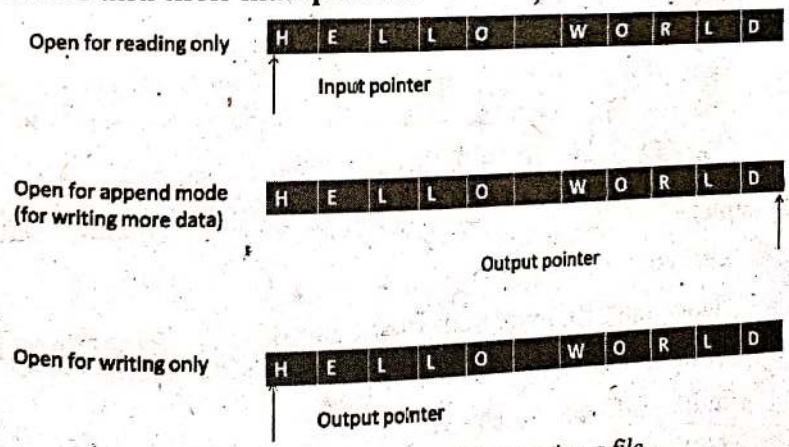


Fig.: Action on file pointers while opening a file

Functions for Manipulation of File Pointers

- The C++ I/O system supports four functions for setting a file pointer to any desired position inside the file or to get the current file pointer.

Function	Member of the class	Action performed
seekg()	ifstream	Moves get file pointer to a specific loacation
seekp()	Ofstream	Moves put file pointer to a specific location
tellg()	ifstream	Returns the current position of the get pointer
tellp()	Ofstream	Return the current position of the put pointer

- Seek functions **seekg( )** and **seekp( )** can also be used with two arguments as follows:

  seekg(offset, refposition);

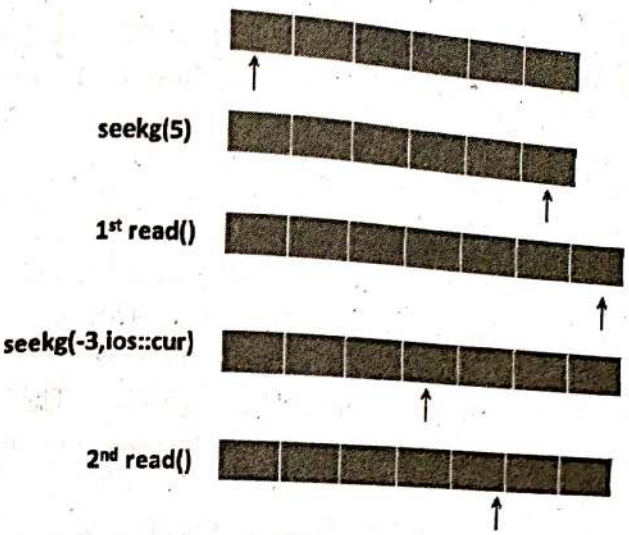
  seekp(offset, refposition);

- The parameter **offset** represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.
- The refposition takesone of the following three constants defined in the ios class:
  - ios::beg start of the file
  - ios::cur current position of the pointer
  - ios::end end of the file

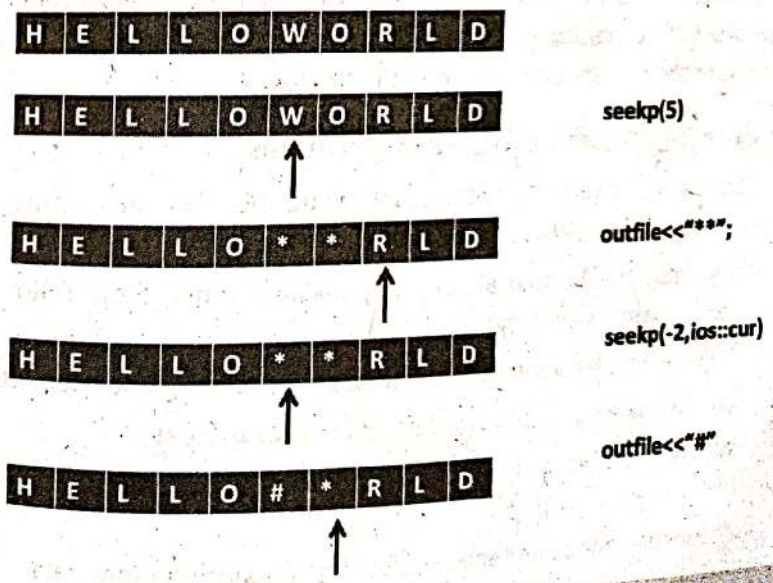
Seek call	Action
fout.seekg(0,ios::beg);	Go to start
fout.seekg(0,ios::cur);	Stay at the current position.
fout.seekg(0,ios::end);	Go to the end of file
fout.seekg(m,ios::beg);	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur);	Go forward by m byte form the current position
fout.seekg(-m,ios::cur;	Go backward by m bytes from the current position
fout.seekg(-m,ios::end);	Go backward by m bytes from the end

Example:

```
char ch;
ifstream infile("myfile.txt", ios::binary);
infile.seekg(5);
infile.read((char*)&ch,sizeof(ch));
infile.seekg(-3, ios::cur);
infile.read((char*)&ch,sizeof(ch));
```




```
ofstream outfile("myfile.txt");
outfile.seekp(5);
outfile<<"**";
outfile.seekp(-2,ios::cur);
outfile<<"#";
```



8.12 Sequential and Random Access to File

Sequential Access to File

In Sequential access, to access a particular data in the file, all data from the start must be accessed and discarded up to that location. The

main disadvantage is all the preceding data must be read. If the desired data is at the end of the storage, all previous data from the top must be accessed. Sequential access is only means of accessing data in some devices like tape drives.

Random Access to File

In random access, to read a data in file preceding data must not be accessed. The data in any desired location can be accessed directly. The random access to file is faster than sequential access. However, the access speed depends on the type of media used to store data. The programmer can specify whether to access sequentially or randomly in storage media like hard disk.

Random access to the file is preferable than sequential because now a days majority of systems that use the random access method also use sequential processing for some portion of the processing activities in the same storage location. Every file maintain two type of pointers called get pointer and put pointer which tell the current position in the file where reading and writing will take place. The pointer helps to implement random access in the file. In sequential access, we need to go through all the record before reaching the specified location, but using get pointer and put pointer we can access any point in the file.

8.13 Testing Errors during File Operations

So far we have been opening and using the files for reading and writing on assumption that everything is fine with files

This may not be true always. For instance, one of the following may happen dealing with files:

1. A file which we are attempting to open for reading does not exist
2. The file name used for a new file may already exist
3. We may attempt an invalid operation such as reading past the end-of-line
4. These may not be any space in the disk for storing more data
5. We may use an invalid file name
6. We may attempt to perform an operation when the file is not opened for that purpose

The class ios supports several member functions that can be used to read the status recorded in a file stream.

SOME EXAMPLES

1. Write a program that will have a class with members, name, rollno, director and filename and other member function as required. All of the objects of that class will share two members directory and filename where the information of the student is stored. **[2066 Jestha]**

Answer:

```
#include<iostream>
using namespace std;
class student
{
    int rollno;
    char name[30];
    static char directory[15];
    static char filename[15];
  public:
    void input( )
          {
  cout<<"\nEnter the name of student:";
          cin>>name;
          cout<<"\nEnter the roll number of student:";
          cin>>rollno;
          }
    void display( )
    {
  cout<<"\nName:"<<name;
  cout<<"\nRollno:"<<rollno;
  cout<<"\nDirectory:"<<directory;
  cout<<"\nFilename:"<<filename;
    }
};
char student::directory[15]="KEC";
char student::filename[15]="2069";
int main( )
{
    student s1,s2;
    s1.input( );
    s2.input( );
```

```
    s1.display( );
    s2.display( );
    return 0;
}
```

2. Write a program to overload the stream operators to read and display the complex numbers. [2066 Jestha]

Answer:

```
#include<iostream>
using namespace std;
class complex
{
   int real;
   int imag;
  public:
   friend istream & operator>>(istream &, complex &);
   friend ostream & operator<<(ostream &, complex &);
};
istream & operator>>(istream &input,complex &c)
{
  cout<<"\nEnter the real part:";
  input>>c.real;
  cout<<"\nEnter the imaginary part:";
  input>>c.imag;
  return input;
}
ostream & operator<<(ostream &output,complex &c)
{
  output<<endl<<c.real<<"+i"<<c.imag;
  return output;
}
int main( )
{
  complex c1;
  cin>>c1;
  cout<<"\nThe complex number is :";
  cout<<c1;
  return 0;
}
```

3. Write an interactive program to maintain student database. The information to be stored in the database is registration number, name, program, contact number and address. The user must be able to access all detail about a student by entering the registration number. [2067 Ashad]

Answer:

```
#include<iostream.h>
#include<fstream.h>
#include<process.h>
using namespace std;
class student
{
        int reg_no;
        char name[40];
        char program[20];
        long int contact_no;
        char address[40];
public:
        void getdata( )
        {
          cout<<endl<<"Enter Registration number of student:";
          cin>>reg_no;
          cout<<endl<<"\nEnter Name of Student:";
          cin>>name;
          cout<<endl<<"\nEnter Address of Student";
          cin>>address;
          cout<<endl<<"\nEnter program of Student:";
          cin>>program;
          cout<<endl<<"\nEnter contact
        number:";cin>>contact_no;
        }
void showdata( )
  {
        cout<<"Registration number:"<<reg_no<<endl;
        cout<<"\nName:"<<name<<endl;
        cout<<"\nAddress:"<<address<<endl;
        cout<<"\nProgram:"<<program<<endl;
        cout<<"\nContact Numbera:"<<contact_no<<endl;
```

```
    }
     int search(int r)
     {
          if(reg_no==r)
               return 1;
          else
               return 0;
     }
     void showRecord( );
     void inputRecord( );
};
void student::showRecord( )
 {
  student pers;
    int Reg,flag=0;
    fstream file;
    file.open("student.txt",ios::in|ios::out|ios::binary);
    cout<<"Enter Registration number:"<<endl;
    cin>>Reg;
    do
    {
                if(pers.search(Reg))
                {
                          pers.showdata( );
                          flag=1;
                          break;
                }
    }while (file.read((char*)&pers,sizeof(pers)));
  if(flag==0)
    cout<<"Not found"<<endl;
    file.close( );
}
void student::inputRecord( )
{
        student pers;
        fstream file;
        file.open("student.txt",ios::in|ios::out|ios::binary|ios::app);
        char ch;
        do
         {
            cout<<"\nEnter student's data:";
            pers.getdata( );
            file.write((char*)&pers,sizeof(pers));
            cout<<"Enter another student information(y/n)?";
            cin>>ch;
        }while(ch=='y' || ch=='Y');
        file.close( );
   }
int main( )
{
        int n;
        student pers;
        while(1)
        {
            cout<<"1. Input Record."<<endl;
            cout<<"2. Search by Registration number."<<endl;
            cout<<"3. Exit."<<endl;
            cin>>n;
            switch(n)
             {
                case 1:
                    pers.inputRecord();
                    break;
                case 2:
                    pers.showRecord();
                    break;
                case 3:
                    exit(0);
                    break;
                default:
                    cout<<"Enter number between 1-3 only";
             }
        }
  return0;
  }
```

4. Create a class student with member variables roll, name, telephone and score. Member functions to get the input from keyboard and display state of object on the screen. Also write two member functions one to store the object in disk file student.dat and other to read the record from the file. Write main function create an object, take details from user and store the details in file. Also, display the record from the file. [2067 Magh]

Answer:

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
class student
{
    char name[40];
    int roll;
    int telephone;
    int score;
  public:
    void getdata( )
    {
        cout<<"\nEnter name of student:";cin>>name;
        cout<<"Enter Roll number:";cin>>roll;
        cout<<"Enter phone number:";cin>>telephone;
        cout<<"Enter score:";cin>>score;
    }
    void showdata( )
    {
        cout<<"\nName of Student:"<<name;
        cout<<"\nRoll number:"<<roll;
        cout<<"\nPhone number:"<<telephone;
        cout<<"\nScore:"<<score;
    }
};
void input_to_file( )
{
    fstream file;
    student s1;
    file.open("student.txt",ios::binary|ios::out|ios::binary);
    cout<<"\nEnter student data:";
    s1.getdata( );
    file.write((char*)&s1,sizeof(s1));
}
void output_from_file( )
{
    fstream fout;
    student s1;
    fout.open("student.txt",ios::out|ios::in|ios::binary);
    fout.seekg(0);
    fout.read((char*)&s1,sizeof(s1));
    while(!fout.eof( ))
    {
        cout<<"\nStudent record:";
        s1.showdata( );
        fout.read((char*)&s1,sizeof(s1));
    }
}
int main( )
{
    input_to_file( );
    output_from_file( );
    return 0;
}
```

5. Write a program to display the output in pyramid form as follows. [2068 Baisakh]

A
AB
ABC
ABCD

Answer:

```
#include<iostream.h>
#include<iomanip.h>
using namespace std;
int main( )
   {
        cout.width(4);
        cout<<"A";
        cout<<endl;
        cout.width(4);
        cout<<"AB";
        cout<<endl;
        cout.width(4);
        cout<<"ABC";
        cout<<endl;
        cout.width(4);
        cout<<"ABCD";
      return 0;
   }
```

6. Write a class student with roll, name, address, marks as member variables. Use a member function to write records of students in a binary file and another member function to read records from file. Write a program to search a specific record of student using roll number as key from user input. [2068 Chaitra]

Answer:

```
#include<iostream>
#include<fstream.h>
#include<process.h>
using namespace std;
class student
{
      int roll;
      char name[40];
      char address[20];
      int marks;
  public:
      void getdata( )
      {
        cout<<endl<<"Enter Roll of Student:";cin>>roll;
        cout<<endl<<"Enter Name of Student:";cin>>name;
      cout<<endl<<"\nEnter Address of Student";cin>>address;
      cout<<endl<<"Enter marks of Student:";cin>>marks;
      }
      void showdata( )
      {
      cout<<"Roll:"<<roll<<endl;
        cout<<"\nName:"<<name<<endl;
      cout<<"\nAddress:"<<address<<endl;
      cout<<"\nMarks:"<<marks<<endl;
      }
    int search(int r)
    {
        if(roll==r)
        return 1;
      else
        return 0;
    }
    void showRecord( );
    void inputRecord( );
};
void student::showRecord( )
{
        student pers;
        int Roll,flag=0;
    fstream file;
    file.open("student.txt",ios::in|ios::out|ios::binary);
        cout<<"Enter Roll number:"<<endl;
    cin>>Roll;
    do
    {
      if(pers.search(Roll))
      {
        pers.showdata( );
        flag=1;
        break;
      }
    }while (file.read((char*)&pers,sizeof(pers)));
```

```
            if(flag==0)
          cout<<"Not found"<<endl;
          file.close( );
}
  void student::inputRecord( )
    {
              student pers;
          fstream file;

  file.open("student.txt",ios::in|ios::out|ios::binary|ios::app);
          char ch;
             do
         {
                    cout<<"\nEnter student's data:";
                    pers.getdata();
                    file.write((char*)&pers,sizeof(pers));
                    cout<<"Enter another student
  information(y/n)?";
                    cin>>ch;
          }while(ch=='y' || ch=='Y');
      file.close();
    }
  int main( )
  {
      int n;
      student pers;
      while(1)
      {
          cout<<"1. Input Record."<<endl;
          cout<<"2. Search by Rollnumber."<<endl;
          cout<<"3. Exit."<<endl;
          cin>>n;
          switch(n)
      {
          case 1:
        pers.inputRecord( );
        break;
          case 2:
       pers.showRecord( );
```



```
        break;
          case 3:
        exit(0);
        break;
          default:
          cout<<"Enter number between 1-3 only";
      }
  }
  return 0;
  }
```

7. Write a program that stores information of a students in a file and display the file's content in descending order according to their marks obtained. **[2070 Ashad]**

Answer:

```
#include<iostream.h>
#include<fstream.h>
using namespace std;
class person
{
    char name[30];
  public:
    int marks;
    void inputdata( )
      {
        cout<<"Enter name:"<<endl;
        cin>>name;
        cout<<"Enter marks:"<<endl;
        cin>>marks;
      }
    void showdata( )
      {
        cout<<"\n Name:"<<name;
        cout<<"\n Marks obtained:"<<marks;
      }
};
int main( )
{
      person pers[3];
```

```
fstream file;
person p[3];
file.open("person1.txt",ios::app|ios::out|ios::in);
for(int i=0;i<3;i++)
{
        pers[i].inputdata( );
        file.write((char*)&pers[i],sizeof(pers[i]));
}
file.seekg(0);
int i = 0;
while(!file.eof( ))
{
        file.read((char*)&p[i],sizeof(p[i]));
        i++;
}
person a;
for(i = 0; i<3;i++)
{
        for(int j = 0; j<3; j++)
        {
        if(p[j].marks>p[j+1].marks)
            {
                a =  p[j];
                p[j] = p[j+1];
                p[j+1]= a;
            }
        }
}
for(i = 0; i<3; i++)
{
 p[i].showdata( );
cout<<endl;
}
return 0;
}
```

8. Write a program to make billing system of a department store. Your program should store and retrieve data to/from files. Use manipulators to display the record in proper formats. **[2072 Chaitra]**

Answer:

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
class department_store
  {
        int trans_id,bill_no;
        char qty1[10],qty2[10];
        unsigned int trans_date;
      public:
        void getdata( )
        {
          cout<<"\nEnter Transaction id no:";
         cin>>trans_id;
          cout<<"\nEnter bill number";
        cin>>bill_no;
          cout<<"\nEnter First Price of Quatity:";
         cin>>qty1;
          cout<<"\nEnter second Price of Quantity:";
         cin>>qty2;
        }
         void displayall(department_store s)
         {
       fstream readFile;
       readFile.open("department.txt",ios::in|ios::out|ios::app);
       readFile.seekg(0,ios::end);   //to check emptyness of file
       int a = readFile.tellp( );
       readFile.seekg(0);
        if(a!=0)
        {
          while(readFile.read((char*)&s,sizeof s))
          s.display( );
        }
       else
       {
          cout<<"~~~~File is empty~~~~\n";
       }
```

```
    }
    void display( )
    {
      cout<<"\n\nRecord of ITEM:\n\n";
      cout.setf(ios::showpoint);
      cout<<setw(15)<<"Transaction ID";
      cout<<setw(15)<<"Bill       Number"<<setw(20)<<"First
          Quantity";
      cout<<setw(20)<<"Second Quantity";
      cout<<endl;
      cout<<setw(15)<<trans_id;
      cout<<setw(15)<<bill_no;
      cout<<setw(20)<<qty1;
      cout<<setw(20)<<qty2;
    }
    void add(department_store);
};
void department_store::add(department_store b)
  {
    fstream fin;
    fin.open("department.txt",ios::app|ios::out);
    cout<<"\nItem record\n";
    b.getdata( );
    fin.write((char*)&b,sizeof(b));
    fin.close( );
  }
int main( )
  {
    department_store b;
    //b.getdata( );
    b.add(b);
    b.displayall(b);
    return 0;
  }
```

9. Write a program to read and write the information of 10 students in a file. Also modify the student information according to the given roll number. **[2073 Shrawan]**

Answer:

```
#include<iostream>
#include<fstream>
#include<process.h>
#include<stdlib.h>
using namespace std;
class student
{
        int roll;
 char name[40];
        char address[20];
 int marks;
  public:
 void getdata( )
 {
 cout<<endl<<"Enter Roll of Student:";cin>>roll;
        cout<<endl<<"Enter Name of Student:";cin>>name;
 cout<<endl<<"\nEnter Address of Student";cin>>address;
 cout<<endl<<"Enter marks of Student:";cin>>marks;
 }
 void showdata( )
 {
 cout<<"Roll:"<<roll<<endl;
        cout<<"\nName:"<<name<<endl;
 cout<<"\nAddress:"<<address<<endl;
 cout<<"\nMarks:"<<marks<<endl;
 }
        int search(int r)
        {
        if(roll==r)
        return 1;
                else
        return 0;
        }
        void showRecord( );
        void inputRecord( );
};
void student::showRecord( )
 {
 student pers;
 int Roll,flag=0;
        fstream file;
```

```
        file.open("student.txt",ios::in|ios::out|ios::binary);
    cout<<"Enter Roll number:"<<endl;
        cin>>Roll;
        do
        {
            if(pers.search(Roll))
            {
        pers.showdata( );
        flag=1;
        break;
            }
        }while (file.read((char*)&pers,sizeof(pers)));
  if(flag==0)
        cout<<"Not found"<<endl;
        file.close( );
  }
 void student::inputRecord( )
  {
      student pers;
      fstream file;
      file.open("student.txt",ios::in|ios::out|ios::binary|ios::app);
      char ch;
      cout<<"\nEnter student information:";
      pers.getdata( );
      file.write((char*)&pers,sizeof(pers));
      file.close( );
  }
int main( )
{
 int n;
 student pers[10],per;
 while(1)
 {
  cout<<"1. Input Record."<<endl;
  cout<<"2. Search by Rollnumber."<<endl;
  cout<<"3. Exit."<<endl;
  cin>>n;
  switch(n)
  {
      case 1:
      for(int i=0;i<10;i++)
       {
          pers[i].inputRecord( );
       }
      break;
      case 2:
          per.showRecord( );
          break;
      case 3:
          exit (0);
          break;
     default:
          cout<<"Enter number between 1-3 only";
    }
 }
 return 0;
}
```

10. Create class student to store Name, Age and CRN of students. Write a program to write records of N numbers of students into the file. And your program should search complete information of students from file according to CRN entered by user and display it. [2073 Shrawan]

Answer:

```
#include<iostream>
#include<fstream>
#include<process.h>
#include<stdlib.h>
using namespace std;
class student
{
    int CRN;
    char name[40];
    int age;
  public:
```

```
    void getdata( )
    {
     cout<<endl<<"Enter Roll of Student:";cin>>CRN;
     cout<<endl<<"Enter Name of Student:";cin>>name;
     cout<<endl<<"Enter Age of Student:";cin>>age;
     }
    void showdata( )
    {
     cout<<"College Roll Number:"<<CRN<<endl;
     cout<<"\nName:"<<name<<endl;
     cout<<"\nAge:"<<age<<endl;
     }
    int search(int r)
     {
   if(CRN==r)
   return 1;
         else
   return 0;
     }
   void showRecord( );
   void inputRecord( );
};
void student::showRecord( )
 {
 student pers;
 int Roll,flag=0;
        fstream file;
        file.open("student.txt",ios::in|ios::out|ios::binary);
cout<<"Enter Roll number:"<<endl;
        cin>>Roll;
        do
        {
```

```
            if(pers.search(Roll))
             {
               pers.showdata( );
               flag=1;
               break;
             }
           }while (file.read((char*)&pers,sizeof(pers)));
  if(flag==0)
          cout<<"Not found"<<endl;
          file.close( );
  }
void student::inputRecord( )
  {
       student pers;
       fstream file;
       file.open("student.txt",ios::in|ios::out|ios::binary|ios::app);
       char ch;
       cout<<"\nEnter student information:";
       pers.getdata( );
       file.write((char*)&pers,sizeof(pers));
      file.close( );
  }
int main( )
{
      int n;
    student pers[20],per;
    while(1)
    {
        cout<<"1. Input Record."<<endl;
        cout<<"2. Search by Rollnumber."<<endl;
        cout<<"3. Exit."<<endl;
        cin>>n;
```

```
        switch(n)
        {
        case 1:
        cout<<"Input number of Record of Student to be included\n";
        cin>>n;
        for(int i=0;i<n;i++)
        {
        pers[i].inputRecord( );
        }
        break;
      case 2:
          per.showRecord( );
        break;
    case 3:
        exit (0);
        break;
    default:
        cout<<"Enter number between 1-3 only";
    }
  }
return 0;
}
```

11. Write a program to make simple library management system of a college. Your program should store an retrieve the information (Book Name, Book ID, Number of books and purchase date). **[2073 Chaitra]**

Answer:

```
#include<iostream>
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
class library
{
  char BookName[20];
  int BookID;
  int No_of_book;
  int day,month,year;
public:
  void input( )
  {
    cout<<"\nEnter the name of Book:";
    cin>>BookName;
    cout<<"\nEnter the BookID:";
    cin>>BookID;
    cout<<"\nEnter the Number of Book:";
    cin>>No_of_book;
    cout<<"\nEnter the purchase day, month and year:";
    cin>>day>>month>>year;
  }
  void display( )
  {
    cout<<"\nBook ID:"<<BookID;
    cout<<"\nBook Name:"<<BookName;
    cout<<"\nNumber of Book:"<<No_of_book;
    cout<<"\nPurchased Date:"<<day<<"/"<<month<<"/"<<year;
  }
  void add( )
  {
    fstream fin;
    library s;
    fin.open("student.txt",ios::app|ios::out|ios::binary);
    cout<<"\nThe library record:";
    s.input( );
    fin.write((char*)&s,sizeof(s));
    fin.close( );
  }
  void displayAll( )
  {
    fstream fout;
    library s;
    fout.open("student.txt",ios::in|ios::binary);
```

```
        while(fout.read((char*)&s,sizeof(s)))
        {
          s.display( );
        }
        fout.close( );
      }
};

int main( )
{
  cout<<"Enter the detail of the book:"<<endl;
  library s1;
  s1.add( );
  s1.displayAll( );
  return 0;
}
```

12. Write a program to store and retrieve the information of patient (Patient_ID, name, address, age and type) in hospital management system. [2074 Ashwin]

Answer:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
class hospital
{
  char PatientName[20];
  int PatientID;
  char address[20];
  int age;
  char type[20];
public:
  void input( )
  {
    cout<<"\nEnter the PatientID:";
    cin>>PatientID;
    cout<<"\nPatient Name:";
    cin>>PatientName;
    cout<<"\nAddress:";
    cin>>address;
    cout<<"\nAge:";
    cin>>age;
    cout<<"\nEnter the type for visiting department:";
    cin>>type;
  }
  void display( )
  {
    cout<<"\nPatient ID:"<<PatientID;
    cout<<"\nPatient Name:"<<PatientName;
    cout<<"\nAddress:"<<address;
    cout<<"\nAge:"<<age;
    cout<<"\nType:"<<type;
  }
  void add( )
  {
    fstream fin;
    hospital s;
    fin.open("hospital.txt",ios::app|ios::out|ios::binary);
    cout<<"\nThe Hospital record:";
    s.input( );
    fin.write((char*)&s,sizeof(s));
    fin.close( );
  }
  void displayAll( )
  {
    fstream fout;
    hospital s;
    fout.open("hospital.txt",ios::in|ios::binary);
    while(fout.read((char*)&s,sizeof(s)))
    {
      s.display( );
    }
    fout.close( );
  }
};

int main( )
```

```
{
  cout<<"Enter the detail of the Patient:"<<endl;
  hospital s1;
  s1.add( );
  s1.displayAll( );
  return 0;
}
```

13. Write a program to write the information of 10 employee in a file. And also display their details in console. [2074 Chaitra]

Answer:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
class employee
{
  char Name[20];
  int ID;
  char Department[20];
  int age;
  long int salary;
public:
  void input( )
  {
    cout<<"\nEnter the ID:";
    cin>>ID;
    cout<<"\nEmployee Name:";
    cin>>Name;
    cout<<"\nDepartment:";
    cin>>Department;
    cout<<"\nAge:";
    cin>>age;
    cout<<"\nEnter salary:";
    cin>>salary;
  }
  void display( )
  {
    cout<<"\nEmployee ID:"<<ID;
    cout<<"\nEmployee Name:"<<Name;
    cout<<"\nDepartment:"<<Department;
    cout<<"\nAge:"<<age;
    cout<<"\nSalary:"<<salary;
  }
  void add( )
  {
    fstream fin;
    employee s;
    fin.open("employee.txt",ios::app|ios::out|ios::binary);
    cout<<"\nThe Employee record:";
    s.input( );
    fin.write((char*)&s,sizeof(s));
    fin.close( );
  }
  void displayAll( )
  {
    fstream fout;
    employee s;
    fout.open("employee.txt",ios::in|ios::binary);
    while(fout.read((char*)&s,sizeof(s)))
    {
      s.display( );
    }
    fout.close( );
  }
};

int main( )
{
  cout<<"Enter the detail of the Employee:"<<endl;
  employee s1[10];
  for(int i=0;i<10;i++)
  {
    s1[i].add( );
  }
  for(int i=0;i<2;i++)
  {
    s1[i].displayAll( );
  }
  return 0;
}
```

14. Write a program to show opening, reading objects from file, checking end of file and closing the file. [2075 Ashwin]

Answer:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
class employee
{
   char Name[20];
   int ID;
   char Department[20];
   int age;
   long int salary;
public:
   void input( )
   {
      cout<<"\nEnter the ID:";
      cin>>ID;
      cout<<"\nEmployee Name:";
      cin>>Name;
      cout<<"\nDepartment:";
      cin>>Department;
      cout<<"\nAge:";
      cin>>age;
      cout<<"\nEnter salary:";
      cin>>salary;
   }
   void display( )
   {
      cout<<"\nEmployee ID:"<<ID;
      cout<<"\nEmployee Name:"<<Name;
      cout<<"\nDepartment:"<<Department;
      cout<<"\nAge:"<<age;
      cout<<"\nSalary:"<<salary;
   }
   void add( )
   {
      fstream fin;
      employee s;
      fin.open("employee.txt",ios::app|ios::out|ios::binary);
      cout<<"\nThe Employee record:";
      s.input( );
      fin.write((char*)&s,sizeof(s));
      fin.close( );
   }
   void displayAll( )
   {
      fstream fout;
      employee s;
      fout.open("employee.txt",ios::in|ios::binary);
      while(!fout.eof( ))
      {
         cout<<endl;
         s.display( );
         fout.read((char*)&s,sizeof(s));
      }
      fout.close( );
   }
};

int main( )
{
   cout<<"Enter the detail of the Employee:"<<endl;
   employee s1;

   s1.add( );
   s1.displayAll( );

   return 0;
}
```

CHAPTER 9

TEMPLATES

An important feature of C++ is template which provides great flexibility to the language. It is one of the most sophisticated, flexible and powerful feature of C++. It was not the original feature of C++ as it was added later on. Template support generic programming, allowing development of reusable software components with function and classes, supporting different data types in a single framework. Because of generic nature of programming, template support programming using types as their parameters.

9.1 Function Templates

Syntax for function template:

```
template<class template_type,...>
return_typefunc_name(parameter_list)
{
  //function body
}
```

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, float but by a name that can stand for any typel

E.g.,

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

template <class T>
T GetMax (T a, T b) {
 T result;
 result = (a>b)? a : b;
 return (result);
}

int main ( ) {
 int i=5, j=6, k;
 long l=10, m=5, n;
 k=GetMax<int>(i,j);
 n=GetMax<long>(l,m);
 cout << k << endl;
 cout << n << endl;
 return 0;
}
```

9.2 Overloading Function Template

9.2.1 Overloading with Functions

```
#include<iostream>
using namespace std;
#include<conio.h>
template <class T>
T find_max (T a, Tb)
{
        return (a>b) ? a : b;
}
char* find_max(char *a, char *b)
{
        char *result;
        if(strcmp(a,b)>0)
                result = a;
        else
                result = b;
        return result;
int main( )
{
        int i=10,j=20;
        cout<<endl<< find_max (i,j);
        float a=3.14,b=-6.28;
        cout<<endl<< find_max (a,b);
        char ch='A' ,dh='Z';
```

```
    cout<<endl<< find_max (ch,dh);
    char str1[]="apple",str2[]="zebra";
    cout<<endl<< find_max (str1,str2);
    return 0;
}
```

9.2.2 Overloading with other Template

```
#include<iostream>
using namespace std;
template <class T>
void display( T data)
{
cout<<data<<endl;
}
template <class T>
void display( T data, int n)
{
for(int i = 0; i<=n; i++)
cout<<data<<endl;
}
int main ( )
{
display(1);
display(1.5);
display(420,2);
display("Let's concentrate! Do not make noise!",3);
return 0;
  }
```

9.3 Class Template

Syntax for class template

```
template<class template_type,....>
classclass_name
{
private:
//data member of template type or non template type
//....
public:
//function members with template type argument and return type
};
```

Once a class template is declared, we create a specific instance of the class using following syntax

```
class_name<data_type> object_name;
```

or

```
class_name<data_type1,data_type2,...,data_typeN>object_name;
```

[for the class using multiple template parametres]

We also have the possibility to write class templates, so that a class can have members that use template parameters as types.

A class that operates on any type of data is called class template.

E.g.,

```
#include <iostream>
using namespace std;
template <class T>
class mypair {
      T a, b;
      public:
   mypair (T first, T second)
    {a=first; b=second;}
   T getmax ( );
};

template <class T>
T mypair<T>::getmax ( )
{
 T retval;
 retval = a>b? a : b;
 return retval;
}
int main ( ) {
  mypair <int> myobject (100, 75);
```

```
    cout << myobject.getmax( );
    return 0;
  }
```

9.3.1 Function Definition of Class Template

We can use more than one generic type in class template. Each type are declared in Template specification, and separated by comma operator. The general form is:

```
template<class T1, class T2,....>
class className
{
  //Body of class
};
```

Defining the object of this class is :

```
Classname<type1, type 2,....>objectname;
```

For defining member function outside the class:

```
template <class T1, class T2,.....>
returntype classname<T1,T2,....>::function (arglist)
{
  //Body of function
}
```

Example:

```
#include<iostream>
using namespace std;
template<class T1,class T2, class T3>
class Record
{
  T1 a;
  T2 b;
  T3 c;
public:
  Record(T1 x, T2 y, T3 z)
  {
    a=x;
    b=y;
    c=z;
  }
  void show( );
};
template<class T1, class T2, class T3>
void Record<T1,T2,T3>::show( )
{
  cout<<endl<<a<<" "<<b<<" and "<<c;
}
int main( )
{
  Record<int,char,double>R1(10,'R',1045.12);
  Record<double,char,int>R2(20.1245,'S',12);
  R1.show( );
  R2.show( );
  return 0;
}
```

9.3.2 Non-Template Type Arguments.

Non Template arguments in creating class template

```
template <class template_type>
class class_name
{
private:
template_type variable_name;
//...
public:
return_type func_name(template_type arg);
//...
};
template<class template_type>
return_type  class_name<template_type>::
func_name(template_type arg)
{
  //body of function template
}
```

9.3.3 Default Arguments with Class Template

Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrate this.

Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrate this.

```
template<class T = int, int  size = 10>
class Array
{
        //...
};
int main( )
{
        //...
        Array<float, 5>ftarr; //float array with size 5
        //...
        Array<double>dbarr; //double array with default size of 10
        //...
        Array<>inarr; //default type int with default size 10
}
```

Example:

```
#include<iostream>
using namespace std;
template<class T=double, int size=4>
class Array
{
  T a[size];
public:
  void input(T *temp)
  {
    for(int i=0;i<size;i++)
    {
      a[i]=temp[i];
    }
  }
  void display( );
};
template<class T, int s>
void Array<T,s>::display()
{
  T sum=0;
  for(int i=0;i<s;i++)
  {
    sum=sum+a[i];
  }
  cout<<"\nThe sum is:"<<sum;
}
int main( )
{
  int temp[]={1,2,3,4,5};
  Array<int,5>Obj1;
  Obj1.input(temp);
  double d[]={10.12,11.23,12.02,23.5};
  Array<>Obj2;
  Obj2.input(d);
  Obj1.display( );
  Obj2.display( );
  return 0;
}
```

Output:

The sum is: 15

he sum is : 56.87

9.4 Derived Class Template

- We can create a derive class which is a non-template class from a base class which is a template class.
- We can create a derive class which is a template class from a base class which is not a template class.
- We can create derived class which is a template class from a base class which is also a template class with the same template parameters as in the base class
- We can create a derive class which is a template class from a base class which is also a template class with additional template parameters in the derived class than that of the class

1. If we do not add extra template parameter and supply the template argument of base class with the data type, we create a non-template derived class as:

```
#include<iosream>
```

```
using namespace std;
template <class T>
class base
{
T data;
public:
base(){}
base(T a){ data = a;}
void display( )
{
    cout<<"data: "<<data<<endl;
}
};
class derived1:public base<int>
{
public:
derived1( ){}
derived1(int a):base<int>(a){}
};
int main( ){
        derived1 obj1(5);
        obj1.display();
}
```

2. If we add extra template parameter in the derived class and supply the template argument of base class with the data type, we create a derived class template as:

```
#include<iostream.h>
using namespace std;
template <class T>
class base
{
T data;
public:
base( ){ }
base(T a){ data = a;}
void display( )
{
        cout<<"data: "<<data<<endl;
}
};
template<class T>
class derived2:public base<int>
{
T data;
public:
derived2( ){ }
derived2(int a, T b):base<int>(a), data(b){ }
void display( ){
cout<<"in base";
base<int>::display( );
cout<<"in derived , data:"<<data<<endl;
}
};
int main( ){
        derived2<float> obj2(10,12.34);
        obj2.display( );
}
```

3. If the base class template parameter is still useful in derived class, the derived class is created as class template i.e base and derived template classes have same template parameter.

```
#include<iostream.h>
using namespace std;
template <class T>
class base
{
T data;
public:
```

```
base( ){}
base(T a){ data = a;}
void display( )
{
        cout<<"data: "<<data<<endl;
}
};
template <class T>
class derived3:public base<T>
{
public:
derived3( ){ }
derived3(T a):base<T>(a){ }
};
int main( ){
        derived3 <int> obj3(5);
        obj3.display( );
}
```

4. We can also add extra template parameter in the derived class along with the base class template parameter.

```
#include<iostream.h>
using namespace std;
template <class T>
class base
{
T data;
public:
base( ){ }
base(T a){ data = a;}
void display( )
{
        cout<<"data: "<<data<<endl;
}
};
template<class T1, class T2>
class derived4:public base<T1>
{
T2 data;
public:
derived4( ){ }
derived4(T1  a, T2  b):base<T1>(a), data(b){}
void display( )
{
cout<<"in base";
base<T1>::display( );
cout<<"in derived , data:"<<data<<endl;
}
};
void main( ){
        derived4<int, float> obj2(10,12.34);
        obj4.display( );
}
```

5. The derived class template can be created from the base class which is not a class template. In this case, a template parameter is added in the derived class during inheritance

```
#include<iostream.h>
using namespace std;
class base
{
int data;
public:
base( ){}
base(int a){ data = a;}
void display( )
{
        cout<<"data: "<<data<<endl;
```

```
}
};
template<class T>
class derived5:public base
{
public:
derived5( ){ }
derived5(int a, T b):base(a),data(b){}
void display( ){
cout<<"in base";
base::display( );
cout<<"in derived , data:"<<data<<endl;
}
};
int main( ){
        derived5<float> obj5(25, 10.5);
        obj5.display( );
}
```

9.5 Introduction to Standard Template Library

The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. The sole purpose of designing STL is to increase performance and flexibility for C++ programmer. The STL is a generic library, meaning that its components are heavily parameterized and almost every component in the STL is a template. So any data type can be used with any algorithm or data structure (class) of library.

The STL contains several components. But in core it contain three fundamental items: **container, algorithms and iterators.**

9.5.1 Containers

1. Sequence Containers

The sequence container is a variable-sized container whose elements are arranged in a strict linear order. It supports insertion and removal of elements. Every data, user defined or built-in, has a specific position in the container.

Sequence Containers store elements in a linear list. Each element is related to one other element by its position along the line. They are divided into following:

a. Vector
b. List
c. Deque

2. Associative Containers

An Associative Container is a variable-sized Container that supports efficient retrieval of elements based on keys. Like Sequence Container, it supports insertion and removal of elements, but differs from a Sequence in that it does not provide a mechanism for inserting an element at a specific position

They are not sequential. There are four types of associative containers:

a. Set
b. Multiset
c. Map
d. Multimap

3. Derived Containers

These container are derived from sequential container. These are also known as container adaptors. The STL provides three derived containers namely,

a. Stack
b. Queue
c. Priority_queue.

Algorithm:

An algorithm is a procedure that is used to process the data contained in the containers. STL algorithm based on the nature of operations they perform may be categorized as:

a. Mutating algorithms
b. Sorting algorithms
c. Set algorithms
d. Relational algorithms
e. Retrieve or not-mutating algorithms

9.5.2 Algorithms

A STL provides algorithms that can be used across a variety of containers, basic data types and even user defined classes. STL algorithms are the global standalone templatized function. STL provides many algorithms that we can frequently use to manipulate containers. Algorithms are used for various types of purposes such as inserting, deleting, searching and sorting.

The algorithms operate on container and array elements indirectly through iterators. Many algorithms operate on sequences of elements defined by pairs of iterators. That is, a first iterator is pointing to the first element of the sequence and a second iterator is pointing to one element past the last element of the sequence.

9.5.3 Iterators

Iterators behave like a pointer and use to access individual elements in container. They are often used to traverse from one element to another, a process known as iterating through the container. That mean if the iterator point to one element in range then it is possible to increase or decrease iterator so that we can access next element in the range. Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers.

Iterators are key factor in bringing generic programming because they are an interface between containers and algorithms. Algorithms usually take iterators as arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write a generic algorithm that operates on many different kinds of containers.

SOME EXAMPLES:

1. Write a program to use vector container from STL. [2066 Jestha]

Answer:

```
#include<iostream>
#include<vector>
using namespace std;
void display(vector <int>&v)
{
for(inti=0;i<v.size();i++)
  cout<<v[i]<<" ";
}
int main( )
{
  system("cls");
vector<int> v;
  cout<<"\nInitial size of vector:"<<v.size( );
  int x;
  cout<<"\nEnter the value to vector:";
  for(inti=0;i<4;i++)
  {
cin>>x;
v.push_back(x);
  }
  cout<<"\nSize after adding 4 values:";
  cout<<v.size( )<<"\n";
  cout<<"\nCurrent Content:";
  display(v);
  return 0;
}
```

2. Create a class template with necessary member variable to represent vectors. Write initialization mechanism and overload the '*' operator to find the scalar product of two vectors. Also write a member function to display the vector. Use above template and write main function to perform scalar product of vectors of type integer and double. [2067 Magh]

Answer:

```
#include<iostream>
using namespace std;
const size=3;
template<class T>
class vector
{
```

```
    private:
        T *v;    //type T vector
    public:
        vector( );
        vector(T *a);
        T operator*(vector &y);
};
template<class T>
vector<T>::vector( )
{
        v=new T[size];
        for(int i=0;i<size;i++)
        v[i] = 0;
}
template<class T>
vector<T>::vector(T *a)
{
        for(int i=0;i<size;i++)
        v[i] = a[i];
}
template<class T>
T vector<T>::operator*(vector &y)
{
        T sum = 0;
for(inti=0;i<size;i++)
        sum+=this->v[i]*y.v[i];
return sum;
}
int main( )
{
        int x[3] = {1,2,3};
        int y[3] = {4,5,6};
        double a[3] = {1.2,3.4,5.6};
        double b[3] = {7.8,8.9,9.1};
        vector<int> v1;
        vector<int> v2;
        v1 = x;
        v2 = y;
        int R1 = v1 * v2;
        vector<double> v3;
        vector<double> v4;
        v3 = a;
        v4 = b;
        int R2 = v3 * v4;
        cout<<"R1="<<R1<<"\n";
        cout<<"R2="<<R2<<"\n";
}
```

3. Write a program with class template to represent array and add member function to find maximum, minimum and sort the generic array. [2068 Baisakh]

Answer:

```
#include<iostream>
using namespace std;
template<class T,int size>
class array
{
  T a[size];
public:
  void input( );
    void display( );
    void assending( );
    T min( );
    T max( );
};
```

```
template<class T, int s>
void array<T,s>::input( )
{
   cout<<"\nEnter element to array of size:"<<s<<"\n";
     for(int i=0;i<s;i++)
   {
     cin>>a[i];
   }
}
template<class T, int s>
void array<T,s>::display( )
{
   cout<<"\nThe array of is:";
     for(int i=0;i<s;i++)
   {
     cout<<a[i]<<"\t";
   }
}
template<class T, int s>
T array<T,s>::min( )
{
   T m=a[0];
     for(int i=1;i<s;i++)
   {
     if(m>a[i])
              {
     m=a[i];
              }
   }
     return m;
}
template<class T, int s>
T array<T,s>::max( )
{
  T l;
  l=a[0];
    for(int i=0;i<s;i++)
  {
    if(l<a[i])
              {
    l=a[i];
              }
  }
    return l;
}
template<class T, int s>
void array<T,s>::assending( )
{
  T temp;
  for(int i=0;i<s-1;i++)
    {
  for(int j=i+1;j<s;j++)
     {
    if(a[j]<a[i])
              {
    temp=a[i];
              a[i]=a[j];
              a[j]=temp;
              }
     }
    }
}
int main( )
{
```

```
    array<int,5>a1;
      a1.input( );
      cout<<"\nMinimum element is:"<<a1.min( );
      cout<<"\nLargest element is:"<<a1.max( );
      a1.assending( );
      a1.display( );
    array<float,5>b1;
      b1.input( );
      cout<<"\nMinimum element is:"<<b1.min( );
      cout<<"\nLargest element is:"<<b1.max( );
      b1.assending( );
      b1.display( );
      return 0;
}
```

4. Write a function template for the function power() which has two parameters base and exp and returns base^{exp}. The type of base is the parameter to the template and exp is int. If the exp is negative, then it must be converted to its positive equivalent. For example, 2^3 and 2^{-3} must both return 8. [2068 Chaitra]

Answer:

```
#include <iostream.h>
#include <math.h>
using namespace std;

template <class T>
T power (T base, T exp)
 {
T result=1;
for(int i=0;i<abs(exp);i++)
 {
    result=result*base;
 }
 return (result);
}
int main ( )
{
 int base=2, exp=8,k;
    int result=power<int>(base,exp);
    cout<<"power("<<base<<","<<exp<<") :"<<result<<endl;
    long l=10, m=5, n;
    k=power<int>(l,m);
    base=3, exp=-3;
    n=power<long>(base,exp);
    cout<<"power("<<l<<","<<m<<") :"<<k<<endl;
    cout<<"power("<<base<<","<<exp<<") :"<<n<<endl;
    return 0;
}
```

5. Write a program to create a stack data structure using class template defining the member functions outside the class template. [2069 Ashad]

Answer:

```
#include<iostream>
using namespace std;
constint MAX = 20;
template<class T>
class Stack
{
    T arr[MAX], top;
  public:
    Stack( );
    void push(T data);
    T pop( );
    int size( );
};
template<class  T>
Stack <T>::Stack( )
{
  top = -1;
}
template<class  T>
void Stack <T>::push(T data)
{
  arr[++top]=data;
```

```
}
template<class T>
T  Stack <T>::pop( )
{
  returnarr[top--];
}
template<class   T>
int   Stack <T>::size( )
{
  return (top+1);
}
int main( )
{
        cout<<"Stack for integers";
        Stack <int>S1;
        cout<<"Size of stack "<<S1.size( )<<endl;
        S1.push(11);
        S1.push(12);
        S1.push(13);
        cout<<"Size of stack "<<S1.size( )<<endl;
        cout<<"Number popped"<<S1.pop( )<<endl;
        cout<<"Number popped"<<S1.pop( )<<endl;
        cout<<"Number popped"<<S1.pop( )<<endl;
        cout<<"Size of stack "<<S1.size( )<<endl;
        cout<<"stack for floats";
        Stack <float>S2;
        cout<<"Size of stack "<<S2.size( )<<endl;
        S2.push(11.5);
        S2.push(12.5);
        S2.push(13.5);
        cout<<"Size of stack "<<S2.size( )<<endl;
        cout<<"Number popped"<<S2.pop( )<<endl;
        cout<<"Number popped"<<S2.pop( )<<endl;
        cout<<"Number popped"<<S2.pop( )<<endl;
        cout<<"Size of stack "<<S2.size( )<<endl;
        return 0;
}
```

6. Write a program that uses a vector container to represent array and display the result in ascending order. For sorting you can use the vector member function. [2069 Ashad]

Answer:

```
#include<iostream>
#include<vector>
using namespace std;
void sort(vector <int>&v) //passing vector element as pass by
                          //reference
{
for(inti=0;i<v.size( );i++)
for(int j=0;j<v.size( );j++)
  {
        int temp=0;
        if(v[i]<v[j])
        {
                temp=v[i];
                v[i]=v[j];
          v[j]=temp;
        }
  }
}
int main( )
{
  vector<int> v;
  v.push_back(11);
        v.push_back(33);
        v.push_back(22);
  v.push_back(2);
  v.push_back(23);
  cout<<"Items in the vector are:"<<endl;
  for(inti=0;i<v.size( );i++)
  {
                cout<<v[i]<<endl;
```

```
        }
    cout<<endl;
    sort(v);  //Vector member function is called for sorting.
        cout<<"After sorting the vector are:"<<endl;
        for(inti=0;i<v.size( );i++)
        {
            cout<<v[i]<<endl;
        }
        return 0;
}
```

7. Write a program to create a stack data structure using class template defining the member functions outside the class template. [2069 Ashad]

Answer:

```
#include<iostream.h>
using namespace std;
const int MAX = 20;
template<class T>
class Stack{
        T arr[MAX], top;
        public:
        Stack( );
        void push(T data);
        T pop( );
        int size( );
};

template <class  T>
Stack <T>::Stack( )
{
        top = -1;
}
template <class  T>
void Stack <T>::push(T data)
{
        arr[++top]=data;
}

template <class T>
T  Stack <T>::pop( )
{
        return arr[top--];
}

template <class  T>
int  Stack <T>::size( )
{
        return (top+1);
}
int main( )
{
cout<<"Stack for integers";
Stack <int>S1;
cout<<"Size of stack "<<S1.size( )<<endl;
S1.push(11);
S1.push(12);
S1.push(13);
cout<<"Size of stack "<<S1.size( )<<endl;
cout<<"Number popped"<<S1.pop( )<<endl;
cout<<"Number popped"<<S1.pop( )<<endl;
cout<<"Number popped"<<S1.pop( )<<endl;
cout<<"Size of stack "<<S1.size( )<<endl;

cout<<"stack for floats";
Stack <float>S2;

cout<<"Size of stack "<<S2.size( )<<endl;

S2.push(11.5);
S2.push(12.5);
S2.push(13.5);

cout<<"Size of stack "<<S2.size( )<<endl;
```

```
cout<<"Number popped"<<S2.pop( )<<endl;
cout<<"Number popped"<<S2.pop( )<<endl;
cout<<"Number popped"<<S2.pop( )<<endl;
cout<<"Size of stack "<<S2.size( )<<endl;

return 0;
}
```

8. Write a program that will find the sum and average of elements in an array using function templates. [2070 Ashad]

Answer:

```
#include<iostream>
using namespace std;
template<class T>
floatsum_arr (T a[ ])
{
        T sum=0;
        float avg;
        for(int i=0;i<10;i++)
        {
                sum+=a[i];
        }
        avg=(float)sum/10;
        cout<<"\nSum in an Array:"<<sum;
        return avg;
}
int main( )
{
        int arr[10];
        for(int i=0;i<10;i++)
        cin>>arr[i];
        float avg=sum_arr(arr);
        cout<<"\nAverage:"<<avg;
        return 0;
}
```

9. Write a program using template to add two numbers. Use the function template to pass integer, float and double. Display the returned result. [2075 Ashwin]

Answer:

```
#include<iostream>
using namespace std;
template <class T1, class T2>
T1 testfunc(T1 a, T2 b){
return (a+b);
}

int main( )
{
int inum1 = 5; float inum2 = 7;
cout<<"\nThe integer sum is"<<testfunc(inum1, inum2);
float fnum1 = 5.6; float fnum2 = 7.5;
cout<<"\nThe float sum is"<<testfunc(fnum1, fnum2);
double dnum1 = 5.6; double dnum2 = 75.67;
cout<<"\nThe double sum is"<<testfunc(dnum1, dnum2);
return 0;
}
```

CHAPTER 10:

EXCEPTION HANDLING

10.1 Error Handling

Two most common types of bugs:

Logic Error: Due to poor understanding of the problem

Syntactic Error: Due to poor understanding of language itself

- We often come across some peculiar problems other than logic and syntax errors. They are known as **exceptions.**
- Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. E.g., **division by zero, access to an array outside its bound, running out of memory or disk space etc.**

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstances" so that appropriate action can be taken.

10.2 Exception Handling Constructs (try, catch, throw)

Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. E.g., division by zero, access to an array outside its bound, running out of memory or disk space etc.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstances" so that appropriate action can be taken.

The exception handling mechanism performs the following task.

1. Find the problem (Hit the exception)
2. Inform that an error has ocurred (throw the exception)
3. Receive the error information(catch the exception)
4. Take corrective actions(handle the exception)

Constructs of exception handling:

The exception mechanism uses three constructs try, throw and catch.

Throw construct:

When a problem is detected during the computation, an exception is raised by using keyword throw. A temporary object of some type is initialized by throw-expression. The type of the object is used in matching the catch block. The syntax of the throw construct is as follows:

throw [obj];

where throw is a keyword and obj may be a named object or nameless object or the argument to throw may not be specified in rethrowing.

Catch Contruct:

The raised exceptions are handled by the catch block. This exception handler is indicated by the keyword catch. The catch construct must be used immediately after the try block. When using multiple catch, other catch blocks must follow the previous catches. Each handler evaluates an exception that matches or can be converted to the type specified in the parameter list if conversion is allowed. The syntax of the catch construct is as follows:

```
catch(type[arg])
{
//exception handling code
}
```

Try Constuct:

The block of code where there is a chance of raising an exception directly from that location or from a function called directly or indirectly from that location is responsible for testing the existence of exceptions. When an exception is raised, the normal program flow is interrupted and the program control is transferred to the appropriate catch block that matches the type of the object thrown as exception. The syntax for the try construct is as follows:

```
try
{
  //code that raise exception or
  //call to a function that raise exception
}
catch(type1 arg)
{
  //action for handling exception for type1
}
catch (type2 arg)
```

```
    //catch for handling exception for type2.
    }
```

Program Code:

```
#include<iostream.h>
int main( )
{
    int a , b;
    cout<<"Enter values of a and b";
    cin>>a>>b;
    int x = a - b;
    try
    {
        if(x!=0)
        cout<<"Result(a/x) = "<<a/x<<"\n";
        else
        throw (x);
    }
    catch(int i)
    {
        cout<<"Divide by zero exception caught";
    }
    cout<<"END";
    return 0;
}
```

10.3 Advantage over Conventional Error Handling

Advantages of Exception handling over Conventional Error handling mechanism:

The errors handling mechanism is somewhat new and very convenient in case of object oriented approach over conventional programming. When error is detected, the error could be handled locally or not locally. Traditionally, when the error is not handled locally the function could

a. Terminate the program.

b. Return a value that indicates error.

c. Return some value and set the program in illegal state.

The exception handling mechanism provides alternatives to these traditional techniques when they are dirty, insufficient and error prone. However, in the absence of exception all these three traditional techniques are used. Exception handling separates the error handling code

from the other code making the program more readable. If the exception is not handled then the program terminates. Exception provides a way for code that detects a problem from which it cannot recover to the part of the code that might take necessary measure.

10.4 Multiple Exception Handling

Multiple catch statements

```
try
{       //try block
}catch(type1 arg)
{       //catch block 1
}catch(type2 arg)
{       //catch block 2
}
...
catch(typeN arg)
{       //catch block N
}
```

Example

```
#include<iostream.h>
using namespace std;
void test(int x)
{
 try
 {
  if(x==1)throw x;
  else if(x==0) throw 'x';
  else  if(x==-1)throw 1.0;
  else if(x==2)throw 1;
  cout<<"End of try-block \n";
 }
 catch(char c)
 {
 cout<<"Caught a character\n";
 }
 catch(int m)
```

```
    {
    cout<<"Caught an integer \n";
    }
    catch(double d)
    {
    cout<<"caught a double\n";
    }
    cout<<"End of try-catch system\n\n";
}
int main( )
{
    cout<<"Testing Multiple Catches\n";
    cout<<"X==1\n";
    test(1);
    cout<<"X==0\n";
    test(0);
    cout<<"X==-1\n";
    test(-1);
    cout<<"X==2\n";
    test(2);
    return 0;
    }
```

10.5 Rethrowing Exception

We can make the handler to rethrow the exception caught without processing it. In such situations, we may simply invoke 'throw' without any argument. This causes the current exception to be thrown to the next enclosing try/catch sequence and caught by a catch statement of that try/catch block.

```
#include<iostream>
using namespace std;
void divide(double x, double y){
cout<<"\n inside the function \n";
try{
if(y == 0.0)
        throw y;          //throwing double
else
        cout<<"Division ="<<x/y<<"\n";
}catch(double){
        cout<<"caught double inside function\n";
        throw;            //rethrowing double
}
cout<<"End of function\n\n";
}
int main( )
{
cout<<"Inside main\n";
try{
        divide(10.5, 2.0);
        divide(10.5, 0.0);
}
catch(double)
{
cout<<"caught double inside main( )\n";
}
cout<<"end of main \n";
return 0;
}
```

10.6 Catching all Exception

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handlers to catch him. In such circumstances, we can force a catch statement to catch all exceptions instead of a certain type alone.

```
catch(...)          //catches all (parenthesis with 3 dots)
{
        //statements for processing all exceptions
}
```

10.7 Exception with Arguments

Sometimes, application may demand more information about the cause of exception.

- With the help of the exception object in parameter, the descriptive information of the cause of the exception can be provided to the exception handler.

Example

```
#include<iostream>
#include<math.h>
#include<string.h>
class NEG
{
      public:
        double value;
        char description[20];
        NEG( )
        {
              value = 0;
              strcpy(description, "\0");
        }
        NEG(double v, char *desc)
        {
              value = v;
              strcpy(description, desc);
         }
};
class Number
{
     double num;
   public:
     void readnum( );
     double sqroot( );
};
void Number::readnum( )
{
  cout<<"Enter number: ";
  cin>>num;
}
double Number::sqroot( )
{
  if(num<0)
  throw NEG(num, "Negative number!!!");
  else
  return sqrt(num);
}
int main( )
{
  Number n1;
    double res;
  n1.readnum( );
  try
    {
    cout<<"\n trying to find out square root ...";
    res = n1.sqroot( );
    cout<<"\n Square root is: "<<res<<endl;
    cout<<"Success... Exception is not reaised"<<endl;
  }
  catch (NEG n)
  {
cout<<"\n Exception occured in square root of"<<n.value<<endl;
cout<<"\n Error! "<<n.description<<endl;
  }
  return 0;
  }
```

10.8 Exception Specification for Function

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition.

The general form of using an exception specification is:

```
return_type  function_name(arg_list) throw (type list)
        {
```

```
        //function body
    }
```

The type list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination.

If we wish to prevent a function from throwing any exception, we may do so bymaking the type list empty

```
return_type function_name(arg_list) throw ( )
    {
        //function body
    }
```

10.9 Handling Uncaught and Unexpected Exceptions

Handling Uncaught Exceptions

If the exception is thrown but not caught, the *terminate( )* function is called. The terminate function will be called when program is attempting to rethrow exception when no exception was originally thrown, when the exception handling mechanism finds the stack corrupted and when a destructor called during stack unwinding caused by an exception tries to exit using an exception. In general, *terminate( )* is the handler of last resort when no other handlers for an exception are available. To change the terminate handler, use **set_terminate** which is defined under exception class.

Example:

```
#include<iostrem>
using namespace std;
void test_handler( )
{       cout<<"\n Inside test handler:";
}
int main( )
{       set_terminate(test_handler);
 try
 {      cout<<"\nInside try block";
        throw 10;
 }
catch(char c)
{       cout<<"\nCharacter exception:";
}
return 0;
}
```

Handling unexpected Exception

Similar to uncatch exception, when a function attempts to throw an exception that is not in the throw list, ***unexpected( )*** function is invoked. By default unexpected( ) call *terminate( )*. Like in terminate( ), we can change the function that is called by unexpected( ). To change unexpected handler, use set_unexpected( ) defined in exception class.

Example:

```
#include<iostream>
using namespace std;
void test_unexpected( )
{  cout<<"\nUnexpected exception:";
}
void calculate (int a, int b) throw (int)
{  cout<<"\n Inside function:";
   if(b==0)
                Throw 'A';
   if(b==1)
                throw 1.0;
   cout<<"\n Quotion is :"<<a/b;
}
int main( )
{
   set_unexpected (test_unexpected);
int x,y;
try
{  calculate (10,1);
}
catch(int c)
{  cout<<"\nInteger Exception:";
}
catch(double d)
{  cout<<"\nDouble Exception:";
}
cout<<"\nEnd of Program:";
return 0;
}
```

SOME EXAMPLES

1. Write a meaningful program illustrating the use of both Exception with argument and Exception specification for function. [2068 Chaitra]

Answer

```
#include<iostream>
#include<math.h>
#include<string.h>
using namespace std;
class NEG
{
    public:
      double value;
      char description[20];
      NEG( )
      {
          value = 0;
          strcpy(description, "\0");
      }
      NEG(double v, char *desc)
      {
          value = v;
          strcpy(description, desc);
      }
};
class Number
{
    double num;
  public:
    void readnum( );
    double sqroot( );
};
void Number::readnum( )
{
  cout<<"Enter number: ";
  cin>>num;
}
double Number::sqroot( )
{
  if(num<0)
  throw NEG(num, "Negative number!!!");
  else
  return sqrt(num);
}
int main( )
{
  Number n1;
    double res;
  n1.readnum( );
  try
    {
    cout<<"\n trying to find out square root ...";
    res = n1.sqroot( );
    cout<<"\n Square root is: "<<res<<endl;
    cout<<"Success... Exception is not reaised"<<endl;
  }
  catch (NEG n)
  {
cout<<"\n Exception occured in square root of"<<n.value<<endl;
cout<<"\n Error! "<<n.description<<endl;
  }
  return 0;
}
```

Program: Exception Specification for function

```
//throw restriction
#include<iostream>
using namespace std;
void test(int x)
throw(int, double, char)
{
    if(x == 0)      throw 'x';
    if(x == 1)      throw x;
    if(x == -1)     throw 1.0;
    cout<<"End of function block \n";
}
int main( )
{
    try
    {
            cout<<"Testing throw restrictions \n";
            cout<<"x = 0\n";test(0);
            cout<<"x = 1\n";test(1);
            cout<<"x = -1\n"; test(-1);
```

```
            cout<<"x = 2\n";test(2);
        }
        catch(char c)
        {
        cout<<"caught a charactor\n";
        }
        catch(int m)
        {
        cout<<"caught an integer\n";
        }
        catch(double d)
        {
        cout<<"caught a double\n";
        }
        cout<<"End of try-catch system\n\n";
        return 0;
    }
```

2. Write a program to find the square root of given number. Check the validity of input number and raise the exception as per requirement. ***[2072 Chaitra]***

Answer:

```
#include<iostream>
#include<cmath>
using namespace std;
class Number
{
   double num;
public:
   class NEG{ };
   void readnum( );
   double sqroot( );
};
void Number::readnum( )
{
        cout<<"Enter number:";
        cin>>num;
}
double Number::sqroot( )
{
  if(num<0)
     throw NEG( );
  else
     return (sqrt(num));

}
int main( )
{
  Number n1;
  double res;
  n1.readnum( );
  try
  {
  cout<<"\nTrying to find square root...";
  res=n1.sqroot( );
  cout<<"\nSquare root is:"<<res<<endl;
  cout<<"Success...Exception is not raised"<<endl;
  }
   catch(Number::NEG)
   {
cout<<"\nSquare root of negative number is not possible!"<<endl;
   }
   return 0;
}
```

3. Write a program to read your Date of Birth and display it. Your program should throw multiple exception for day, month and other values not in range using exception class and each exception is handled by separate handler. **[2073 Shrawan]**

Answer:

```
#include<iostream>
using namespace std;
class DOB
{
    int day,month;
  public:
    DOB( )
    {
    day=month=0;
    }
    DOB(int m,int d)
    {
    month=m;
    day=d;
```

```
    class WrongMonth{ };
    class WrongDay{ };
    void readdata( )
    {
          cout<<"\nEnter month:";
          cin>>month;
          cout<<"\nEnter Day:";
          cin>>day;
    }
    void check(DOB d)
    {
      if(d.month>12)
        throw WrongMonth( );
      else if(d.day>30)
        throw WrongDay( );
    }
    void display( )
    {
          cout<<"\nMonth:\t\t"<<month<<endl;
          cout<<"\nDay:\t\t"<<day;
    }
};
int main( )
{
  DOB dob1,dob2;
  dob2.readdata( );
  try
  {
    dob1.check(dob2);
    dob2.display( );
  }
  catch(DOB::WrongMonth)
  {
        cout<<"\nThe month should not exceed 12\n";
  }
  catch(DOB::WrongDay)
  {
        cout<<"\nThe day should not exceed 30\n";
  }
  return 0;
}
```

❑❑❑