Object Oriented Programming in C++

(10 + 2 Text Book)

Punjab School Education Board

Shaibzada Ajit Singh Nagar

Writer:
Mrs. Pooja Arora, Govt Sen. Sec. School Sahauran, SAS Nagar
Mrs. Sukhwinder Kaur, Govt Sen. Sec. School Sahauran, SAS Nagar

Translator:
Mrs. Shaminder Kaur, Govt Sen Sec School, Manuli, SAS Nagar

Vetter:
Mrs. Sukhwinder Kaur, Govt Sen. Sec. School Sahauran, SAS Nagar

Coordinator:
S.Manvinder Singh, Punjab School Education Board

Published by : Secretary, Punjab School Education Board, Vidya Bhavan Phase-8 Sahibzada Ajit Singh Nagar-160062

ਮੁੱਖ ਬੰਧ

ਪੰਜਾਬ ਸਕੂਲ ਸਿੱਖਿਆ ਬੋਰਡ, ਰਾਜ ਦੀ ਸਕੂਲ-ਸਿੱਖਿਆ ਨੂੰ ਸਮੇਂ ਦੀਆਂ ਲੋੜਾਂ ਤੇ ਵੰਗਾਰਾਂ ਦੇ ਪ੍ਰਸੰਗ ਵਿੱਚ ਢਾਲਣ ਤੇ ਨਵਿਆਉਣ ਲਈ ਨਿਰੰਤਰ ਯਤਨਸ਼ੀਲ ਹੈ।

ਅਸੀਂ ਇਤਿਹਾਸ ਦੇ ਉਸ ਕਾਲ-ਖੰਡ ਵਿੱਚੋਂ ਗੁਜ਼ਰ ਰਹੇ ਹਾਂ ਜਿੱਥੇ ਤੇਜ਼ੀ ਨਾਲ ਪਰਿਵਰਤਨ ਵਾਪਰ ਰਹੇ ਹਨ। ਵਿਕਾਸ ਦੀ ਤੋਰ ਤਿਖੇਰੀ ਹੈ। ਇਸ ਵਿਕਸਤ ਸੰਸਾਰ ਨਾਲ ਇਕਸੁਰ ਹੋਣ ਲਈ, ਜਿੱਥੇ ਗਿਆਨ ਦੀਆਂ ਤੰਦਾ ਵਿਸਤਰਿਤ ਹੋ ਗਈਆਂ ਹਨ, ਸੂਚਨਾ ਤਕਨਾਲਜੀ ਤੇ ਕੰਪਿਊਟਰ-ਸਿੱਖਿਆ ਨੂੰ ਸਿੱਖਿਆ ਦਾ ਅਹਿਮ ਅੰਗ ਬਣਾਉਣਾ ਜ਼ਰੂਰੀ ਹੋ ਗਿਆ ਹੈ।

ਇਸੇ ਮਨੋਰਥ ਨਾਲ ਕੰਪਿਊਟਰ-ਤਕਨੀਕ ਦੀ ਸਿਖਲਾਈ ਹਿੱਤ ਇਹ ਪਾਠ-ਪੁਸਕਤ ਤਿਆਰ ਕੀਤੀ ਗਈ ਹੈ ਤੇ ਇਸ ਨੂੰ ਕੰਪਿਊਟਰ ਦੀ ਵੈੱਬਸਾਈਟ 'ਤੇ ਉਪਲੱਭਧ ਕਰਵਾਉਣ ਦਾ ਉਪਰਾਲਾ ਕੀਤਾ ਗਿਆ ਹੈ। ਨਿਸ਼ਚੇ ਹੀ ਇਹ ਸੁਵਿਧਾ ਕੰਪਿਊਟਰ ਸਾਇੰਸ, ਵੋਕੇਸ਼ਨਲ ਗਰੁੱਪ ਦੇ ਵਿਦਿਆਰਥੀਆਂ ਲਈ ਲਾਹੇਵੰਦ ਤੇ ਰੋਚਕ ਸਾਬਿਤ ਹੋਵੇਗੀ।

ਇਹ ਪਾਠ-ਸਮੱਗਰੀ ਕੰਪਿਊਟਰ ਸਿੱਖਿਆ ਦੇ ਵਿਦਵਾਨਾਂ, ਤਜਰਬੇਕਾਰ ਅਧਿਆਪਕਾਂ ਅਤੇ ਸਿੱਖਿਆ ਬੋਰਡ ਦੇ ਵਿਸ਼ਾ-ਮਾਹਿਰਾਂ ਦੇ ਸਾਂਝੇ ਉੱਦਮ ਸਦਕਾ ਤਿਆਰ ਕੀਤੀ ਗਈ ਹੈ। ਇਸ ਲਈ ਖੇਤਰ ਦੇ ਵਿਦਵਾਨ ਤੇ ਸਹਿਯੋਗੀ ਅਧਿਆਪਕ ਸਾਡੇ ਧੰਨਵਾਦ ਦੇ ਪਾਤਰ ਹਨ। ਇਸ ਨੂੰ ਹੋਰ ਬਿਹਤਰ, ਹੋਰ ਉਪਯੋਗੀ ਤੇ ਹੋਰ ਸੰਚਾਰਮਈ ਬਣਾਉਣ ਲਈ ਖੇਤਰ ਵਿੱਚੋਂ ਆਏ ਮੁੱਲਵਾਨ ਸੁਝਾਵਾਂ ਦਾ ਸਦਾ ਸਵਾਗਤ ਹੈ।

ਚੇਅਰਪਰਸਨ

ਪੰਜਾਬ ਸਕੂਲ ਸਿੱਖਿਆ ਬੋਰਡ

ਬਾਰਵੀਂ ਸ਼੍ਰੇਣੀ ਦੇ ਵੋਕੇਸ਼ਨਲ ਗਰੁਪ ਦੀ ਪਾਠ ਪੁਸਤਕ "**Object Oriented Programming in C++**" ਅੰਗਰੇਜੀ ਮਾਧਿਅਮ ਵਿੱਚ ਪਹਿਲੀ ਵਾਰ ਵਿਦਿਆਰਥੀਆਂ ਦੇ ਹਿੱਤ ਲਈ ਤਿਆਰ ਕੀਤੀ ਗਈ ਹੈ। ਵਿਦਿਆਰਥੀਆਂ ਦੀ ਗਿਣਤੀ ਘੱਟ ਹੋਣ ਦੇ ਬਾਵਜੂਦ ਇਹ ਪੁਸਤਕ ਤਿਆਰ ਕਰਕੇ ਵਿਦਿਆਰਥੀਆਂ ਲਈ ਬੋਰਡ ਦੀ ਵੈੱਬ ਸਾਈਟ http://www.pseb.ac.in ਤੇ ਅਪਲੋਡ ਕੀਤਾ ਜਾ ਰਿਹਾ ਹੈ। ਇਹ ਪਸੁਤਕ ਵਿਦਿਆਰਥੀਆਂ ਲਈ ਬਹੁਤ ਲਾਹੇਵੰਦ ਹੋਵੇਗੀ। ਇਸ ਪੁਸਤਕ ਦੀ ਬੇਹਤਰੀ ਲਈ ਖੇਤਰ ਵਿੱਚੋਂ ਆਏ ਸੁਝਾਵਾਂ ਦਾ ਸਤਿਕਾਰ ਕੀਤਾ ਜਾਵੇਗਾ । ਇਸ ਪੁਸਤਕ ਦੀ ਬਿਹਤਰੀ ਸਬੰਧੀ ਸੁਝਾਅ mathoda1@yahoo.com ਤੇ ਭੇਜੇ ਜਾ ਸਕਦੇ ਹਨ।

ਮਨਵਿੰਦਰ ਸਿੰਘ (ਕੋਆਰਡੀਨੇਟਰ)

Index

Lesson 1

OOP Concept

In this chapter, we will able to understand the concept of software quality factors (Internal and External), Genercity and overloading, object oriented philosophy, object oriented design and top down Design.

1.1 Software Quality Factors

We can put stress on software quality factors using object oriented technique, (internal and external) using object oriented technique.

1.1.1 External factor: - These are visible to end users.

For example

* Correctness
* Robustness
* Extendibility
* Reusability
* Compatibility
* Efficiency
* Portability
* Ease of Use
* Functionality

1.1.2 **Internal factor :-** They are visible to software developers only.

They are

* Modularity
* Flexibility
* Reusability

Internal factors are received by computer expert. External factors are related to each other. Undoubtedly they are visible to users but External factors are dependent on internal Quality factors.

1.2 Major External factors :-

1.2.1 **Correctness.** : The ability of software products to perform their tasks, as defined by their specification

* To achieve Correctness, there should be assured specification.

* Correctness is permanent : If lower layer is correct, which means if it start performing according to its specification, then it is surely achieved at end.

1.2.2 **Robustness.** Robustness is the ability of software system to work efficiently in abnormal conditions.

Robustness characterizes what is happening outside.

1.2.3 **Extendibility.** – Is that quality of software system according to which software easily adapts itself to changes of specifications.

1.2.4 **Reusability.** The ability of software elements to serve for the construction of many different applications.

1.2.5 **Compatibility.** Is the ease of combining software elements with others elements is known as compatibility.

1.2.6 **Portability.** The ease of transferring software products to various hardware and software environments.

1.2.7 **Efficiency.** The ability of a software system to place as few demands as possible on hardware while working.

1.2.8 **Ease of Use.** This is the quality of s/w system with which user learn to use s/w easily & find solutions to the problems using software.

1.3 Major Internal factors

1.3.1 **Modularity:-** In modularity program is divide into smaller units or modules for e.g. it is used to make functions in C. We can easily write long Program by dividing into modules. And also can write solutions to complex problems ease. It gives a beautiful and easy view to our program, actually main advantage of modularity is its Reusability. When we divides a complex program into different & meller modules, it can be used in different ways. When we face same problem in another program, then these modules can be are used for them also.

1.3.2 **Reusability:-** Many programmers can use ones written class of data. It is a good time saving technique also. Beside a programmer according to its need can add more features to existing class.

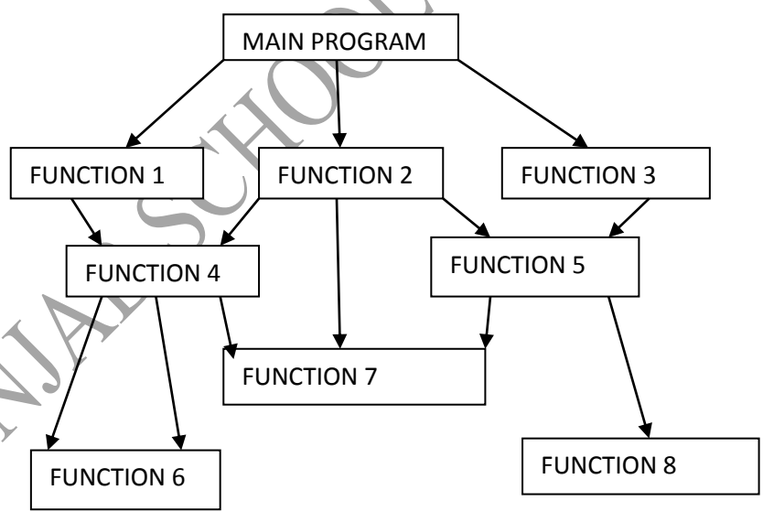
1.4 **Generosity and Overloading:-** Generosity is the prime objective of OOP Generic programs are written and compiled once. But can be used with different data types. We will learn more about her Generosity in lesson – 7. .

Overloading :- We will explain it in next chapter with an operator. If we define one function name more than one or define one operator more one then we do function or operator overloading.

Object Oriented Programming Philosophy:-

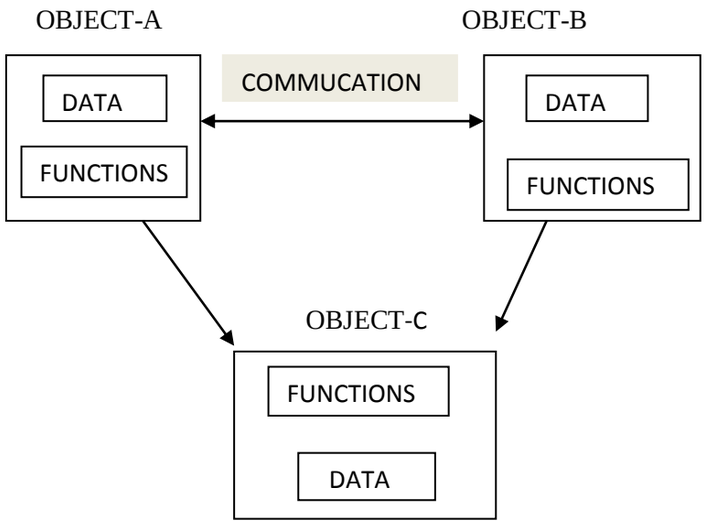
1.5 The term "philosophy" means to have a deep knowledge of something understanding or information. OOP (Object Oriented Programming) : What does oop mean ? We know its full form, but what is it all about? It's a programming style which is based on objects in the real world. That is how it got its name OOP. In General when we think of programming then, we on think about the functionality of a program. Like one program calculates the degree of students . So we usually divide the program based on its functionality. We might build a function to calculate marks of student's marks. Then we can make another's function to determine the total marks, and then we also can create a function to determine the grades of student. . And this process will continues. This kind of habit affects our mind so that we cannot think for another perspective. These symptoms are usually seen in procedural language. Programs i.e. Pascal / C programmers.

Now, we will see how does OOP work ? First of all, we determine the objective of the program. We find out, in the objects rather than in terms of functionality. What we want to achieve from a program we keep out thinking based on object rather than on functionality.



Typical Structure of Procedure-Oriented Programs

OOP treats data as Critical Elements in the program development and does not allow it to flow freely around the system OOP allows us to decompose a program into a number of Entities, called objects and then create data and functions primarily. keeping these objects in mind. The functions in object oriented programs is shown in figure given below:-



Organization of Data and Function in OOP

1.6 Basic Concept of OOP

OOP includes the following basic Concepts: —

1. Objects
2. Classes
3. Data Abstraction and Encapsulation
4. Inheritance
5. Polymorphism
6. Dynamic Binding
7. Message Passing

We shall discuss these concepts in detail in next lessons.

1.7 Top-Down Design and Functional Decomposition)

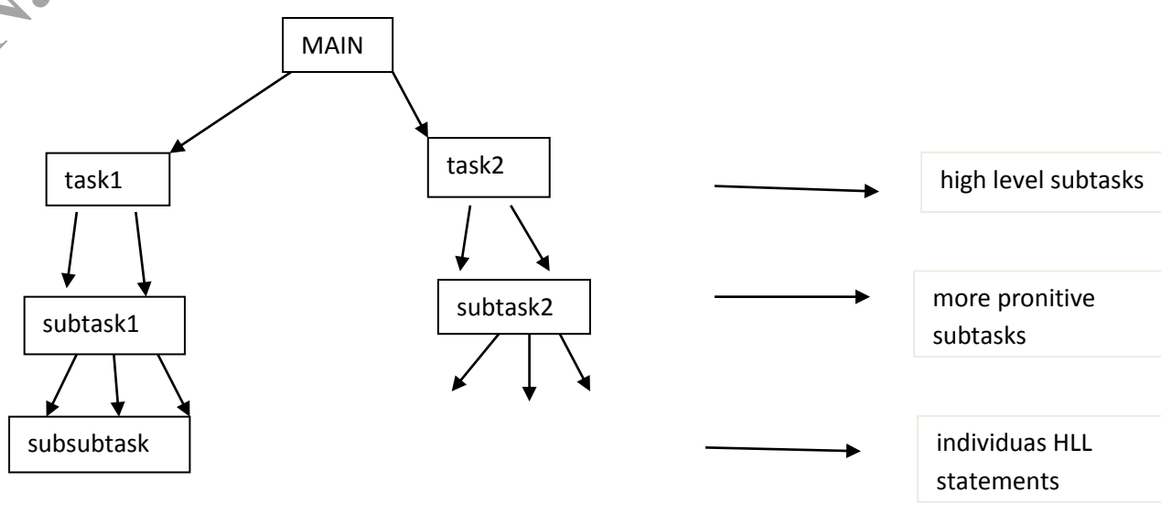
1 Show a problem into high level sub problems.

2 For every high-level Sub-problem.

2.1 Decompose this high- level problem into more simpler Sub-problem.

3 This process should continue till all the Sub-problems get changed in , to programming statements.

Note: Each Sub-task becomes a Sub-program



Points to be Remember:

1. The software Quality factor are of two types external and internal.
2. External factors can be viewed by end users.
3. Internal factors can be viewed by software developer.
4. There should be specified definition to achieve correctness.
5. Program is divided into smaller units or modules in modularity.
6. Object-Oriented programming is a style is based on objects in the real world.

Exercise

1. ***Fill in the blanks :-***

1. …… are viewed by software developer.
2. …… is that when software easily adopts itself to change of specification.
3. ……is dividing the program into smaller units.
4. OOP means……।
5. OOP divides a program into no. of Entities called……

2. Write the answers of Questions :-

1. Write a brief note on Correctness?
2. What is modularity?
3. What is the difference between object oriented programming and procedural programming?
4. What are the basic concepts of OOP ?

Lesson - 2
C++ Language Features

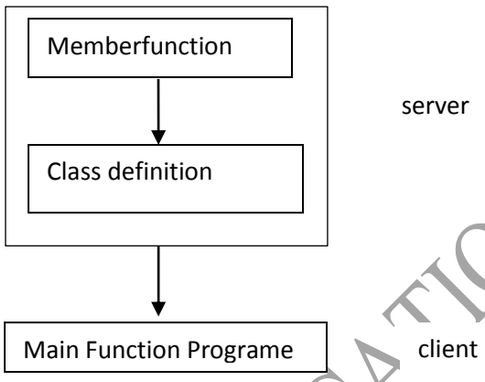
C++ language is an object Oriented Language. It is first developed by Bjarne stroustrup at (AT and T Bell laboratories) situated at New Jersey in USA 1980. Stroustrup wanted to develop a powerful language by combining simula 67 and C language which takes the features of Object Oriented Language along with keeping power & elegance of C language. That is how C ++ language is developed:

In fact, class was the biggest addition to therefore. It is first named " C with classes". In 1983, Its name changed to C++. The thought of name C++ came from C++ increment operator, that recalls C++ is an increment of C.

C++ program has four parts as shown in figure given below

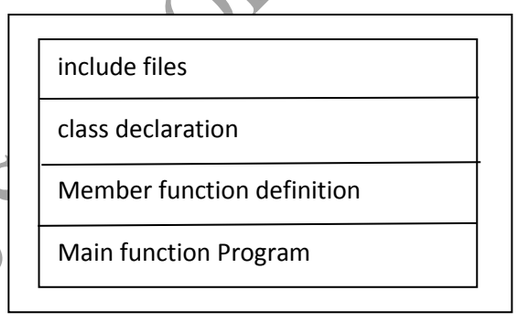
2.1 Structure of C++

C++ program has four parts as shown in figure given below



THE CLIENT-SERVER MODEL

The Client-Server Model
ਚਿੱਤਰ 2.1



Structure of a C++ Program
Diagram 2.1.1

2.2 Input operator

```
# include <iostream.h>
int main( )
{
    float number 1, number 2,
    sum, average ;
cout << "Enter two numbers :"; // prompt
cin >> number 1 ; // Reads numbers
cin >> number 2 ; // from keyboard
sum = number 1 + number 2 ;
average = sum/2 ;
```

```
cout << " sum = " << sum << "\n" ;
cout << "Average =" << average << "\n" ;
return 0 ;
}
```

The output of the program is
Enter two numbers : 6.5 7.5
Sum = 14
Average = 7

2.2.1 cin >> number 1 : is an input statement in which program is said to wait for user to input a number. number 1 is a variable in this statement and c in is predefined object in C++, which is related to standard input stream.

Here this stream specifies the keyboard operator >> is called **extraction** or **get from** operator.

2.3 Output operator

```
# include <iostream.h>
int main( )
{
    cout << "C++ is better than C" ;                    // C++ statement
    return 0;
}          // End of example.
```

There is only one output statement in the Program :
cout << "C++ is better than C." ;
This statement displays string written in quotation marks. In this statement we see two new feature of C++.

cout and <<
cout, is an predefined object C++ . << is called i**nsertion** or **Put to** operator.

2.4 Variable Declaration

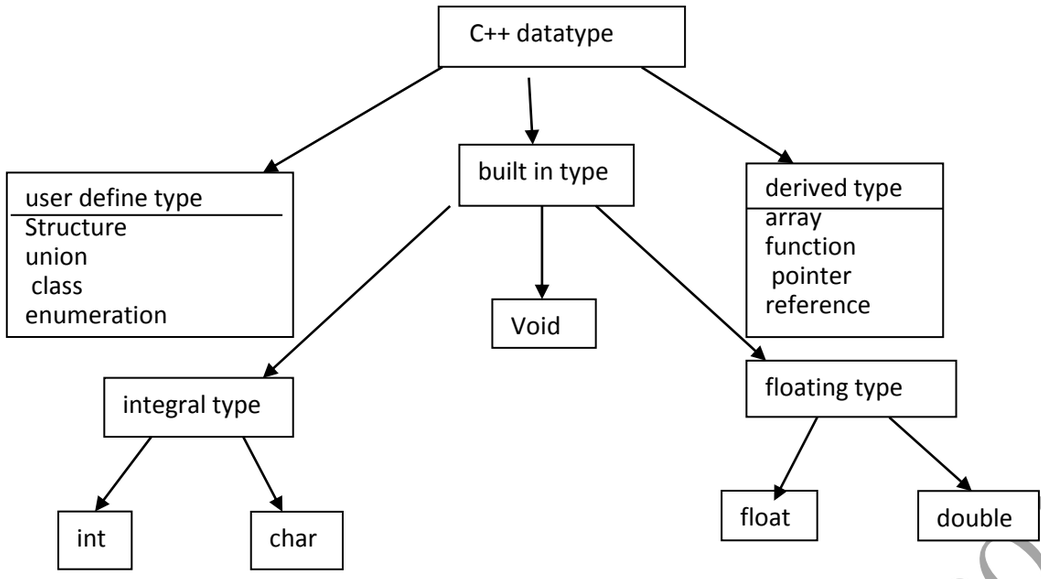
It is necessary to declare all the variables in C++. All the variables are declared at the beginning in C (Before executable statements) but C++ allows to declare its variables anywhere which means variables can be declared where it is to be used.

2.4 **Take an example**

```
int main( )
{
    float x ;
    float sum = 0 ;              // declaration
    for (int i = 1 ; i < 5 ; i++)
    {                            // declaration
    cin >> x ;
    sum = sum + x ;
    }
    float average ;
    average = sum / i ;       //declaration
    cout << average ;
    return 0 ;
}
```

2.5 Data system in C++

Basic data types of C++ are as shown in fig 2.5



C and C++ compiler support all built in data- types.

2.6 C++ Operator

All operator of C are available in C++. Besides it some new. Operator are added also. We have already read about two operators. Insertion operator >> and extraction operator << some more operators are:-

::	Scope resolution operator
::*	Pointer-to-member declarator
→.*	Pointer-to-member operator
* .	Pointer-to-member operator
delete	Memory release operator
endl	Line Feed operator
new	Memory allocation operator
set w	Field width operator

2.7 The const Qualifier

If there is keyword const, it is written before the data type of variable. It shows that the value of variable does not change in the whole program. If the value of variable that is declared with qualifier is changed, error message is displayed by compiler, constants **# defined** are used to change constant.

Const Qualifier assures, that your program variable which is constant does not alter. It also reminds that if anyone is reading the program it is listed that variable cannot changed.

Variable of this qualifier is named in upper case and reminds that these are constants. The programme given below shows the use of const.

```
# include <iostream.h>
void main( )
{
    float r, a ;
    const float Pi = 3.14 ;
    cin >> r ;
    a = Pi * r * r ;
    cout << end 1 << ‘‘Area of Circle = ’’ << a ;
}
```

Const is the better way to use than # define, but its operation is controlled inside or outside the function. If const is placed inside function then its effect will be localized. If it is placed outside then its effect will be global.

2.8 Inline Functions

Inline Functions : The main advantage to use function is that it helps in saving memory space. Only one code is executed while calling function and function body is not made duplicate in memory. When a small function is called several times in a programs, some overheads occur at the time of calling of function.
Time has to be given to pass value, passing control, to return value and to return control.

It means wherever function will be called in source file, actual code will be inserted from function instead of jumping of function. These functions are called inline functions.

The program 2.8 given below shows operation of inline function.

```
# include < iostream.h >
inline void reporters (char * str)
{
    cout << end l << str ;
    exit (1) ;
}
void main ( )
{
                                    // code to open source file
if (file open is failed)
reporter (‘‘unable to open source file’’) ;
                                  // code to open target file
if (file opening failed)
reporters / unable to open target file’’);
                                    //code to copy contents of source file into target file
}
```

Inline functions are similar to #define macro although they give better type checking and neither they have any bad effect. When they are related to macros.

2.9 For example the program given below :

```
# include < iostream.h >
# define SQUARE (X) x*x
inline float square (float y)
{
    return y * y ;
}
void main( )
{
    float a = 0.5, b = 0.5, c, d ;
    c = SQUARE (++ a) ;
    d = Square (++ b) ;
}
```

macro SQUARE gets expanded at the time of preprocessing C = ++x * ++x ; you will note that when macro is expanded, It gives birth to many unexpected results.

2.10 Functions

A function is a set of given instructions in a program which is made to perform a specific task. Every C++ program contains at least one function called main( )

We can divide our program code into different functions. A function is executed only when it is called somewhere in the program. Format of function is given below.

```
type name Parameter 1, Parameter 2, ....)
{
    statements ;
    ..... ;
}
```

where 1 type tells which data type function will return.

2. . name is name of function

3. Parameters are those variables which are going to be passed into function.

4. Statement are those set of instructions for which function is built.

2.10.1 Function Prototypes

Function prototype is a declaration in which arguments to be passed and type value to be return by function, both are defined.

If we have any function foot () to call which receives float and takes arguments as int and returns double value, then its prototype is as follows.

double float(int,int);

Let us see, how functions are defined in C++, some C++ compiler still accepts K & R style. Both formats as given below.

```
                    // K & R style
double foot (a, b)
int a ; float b ;
{
    // some code
}
    // Prototype - Like style
    double foot (int a, float b)
 {
    // Some code
}
```

Note : some C++ compiler accepts both styles.

2.10.2 Function Overloading

In C++, One more thing is added to functions capacity called function overloading. With this facility we see many functions having one name But in C each function is defined with different name in program for e.g. In C, there are many types of functions which return absolute value though they should have different function name there is a different function for every numeric data type. Therefore there are three different functions that return absolute value arguments.

```
int abs (int i) ;
long labs (long l) ;
double fabs (double d) ;
```

Above given functions are doing same work, so it is not necessary to take three different functions. C++ solves this problem and allows three different functions with one name for program. This process is known as function overloading.

This can be explained through the program 2.10.2 given below.

```
# include < iostream.h >
void main( )
{
    int i = – 25, j ;
    long 1 = – 100000 L, m ;
    double d = – 12.34, e ;
   int abs (int) ;
    long abs (long) ;
    double abs (double) ;
    i = abs (i) ;
    m = abs (l) ;
    e = abs (d) ;
}
cout << end 1 << j << end 1 << m << end 1 <<e ;
}
    int abs (int ii)
{
    return (ii > 0 ? ii : ii * –1) ;
```

```
long abs (logn ll)
{
    return (|| > 0 ? || : || * –1) ;
}
    double abs (double dd)
{
    return (dd > 0 ? dd : dd * –1) ;
}
```

how does C++ compiler come to know that which abs( )s will be called ? This Type of argument is passed during call of function.

for example - if int will be passed then integer type is called as abs(gets). If double will be passed then double version of abs(gets) will be called.

2.11 Default arguments in a function

If function is defined to receive two arguments in C, when we call function, we have to pass two values in this function. If we pass one value, then garbage values are considered to be the last argument. When function is called, C++ function has the ability to define those default values for the arguments which are not passed when function is called.

It will be clear from this example.

```
# include < iostream.h >
# include < conio.h >
void box (int sr = 1, int sc = 1, int er = 24, int ec = 80) ;
void main( )
{
    clrscr ( );
    box (10, 20, 22, 70) ;
    box (10, 20, 15) ;
    box (5, 10) ;
    box ( ) ;
}
void box (int sr, int sc, int er, int ec)
{
    int r, c ;
    goto xy (sc, sr)
cout << (char) 218 ; // outputs a graphic character whose ASCII value is 128.
goto xy (ec, sr) ;
cout << (char) 191 ;
goto xy (sc, er) ;
cout << (char) 192 ;
goto xy (ec, er) ;
cout << (char) 217 ;
for (r = sr + 1 ; r < er ; r ++)
{
    goto xy (sc, r) ;
    cout << (char) 179 ;
    goto xy (ec, r) ;
    cout << (char) 179 ;
}
    for (c = sc + 1 ; c < ec ; c++)
  {
    gotoxy (c, sr) ;
    cout << (char) 196 ;
    gotoxy (c, er) ;
    cout << (char) 196 ;
  }
}
```

when we call function box( ) with 4 arguments, then box argument is received when passed, But when we call it with 3 arguments then default value, prototype of box () is used when it is called 2 arguments for last argument, default values are used for last 2 arguments and In the end we call it without argument and a box is drawn which contains all the default values.

2.12 Extern ‘‘C’’ Declaration

In C programming, external variable is that variable which is declared outside the function block. Variable declared inside the function block is called local variable.

An external variable can be accessed by all the functions in the program. It is a global variable. When program begins, memory is allocated to global variable and when program ends it releases memory location. Its life time period is equal to life period time of program.

format file 1 :

```
int Global Variable ;   // implicit definition
void Some Function (void) ;
    // function prototype (declaration)
int main( ) {
Global Variable = 1 ;
Some Function( );
return 0 ;
}
```

file 2 :

```
extern int Global Variable ;  // Explicit declaration
void Some Function (void) {   // Function header (definition)
++ Global Variable ;
}
```

2.13 Reference vs. Pointer

It can be explained by some pointers given below :

1. We can re-assigned a pointer anytime, but reference can not be re-assigned after once initialized.
2. A pointer can point to a NULL but a reference never points to NULL.
3. We cannot get address of reference as in case of pointer.
4. There is no Reference is arithmetic, But we can get address of that object to which reference points and can apply Pointer arithmetic on that object.

2.14 Memory Allocation using New operator and Deallocation using Delete operator

‘C’ uses malloc( ) and calloc( ) functions to allocate memory by run time and dynamic methods. It Also uses free( ) function to free memory that is allocated dynamic. We use dynamic memory technique when it is not clear how much memory space is required. Although C++ supports these functions, but to do memory allocation more efficiently it defines two new operators i.e. unary and delete. New and delete operators do memory allocation more efficiently and in easier way.

An object is created by using new operator and can be destroyed by using delete operator as required. The object created by new operator exists till it is destroyed by delete operator externally. Therefore the control of life time of an object remained directly in our hands and. It does not have any connection with the block structure of program new operator can be used in following way.

Pointer_ Variable = new data_type ;

New operator, allocates memory for data object as required and returns address of object. Data type can be any valid data type. Pointer variable keeps the address of allocated memory space.

example 2.14

p = new float ;
q = new int ;

where p, is pointer of float type and q is pointer of int type.

we can initialize the memory by using new operator for e.g.

Pointer _ Variable = new data_type (Value)

int * *p*= new int (25)
float * *q* = new float (7.5) ;

We can use new operator to create any memory space for any data (also for user defined datatypes) for example arrays, structure and classes. To create memory space for one-dimensional array, following method can be used :-

```
int * p = new int [10]
array_ptr = new int [3] [5] [4] ;     //legal [for multidimensional array]
array_ptr = new int [m] [5] [4] ;     //legal
array_ptr = new int [3] [5] [ ] ;     //illegal
array_ptr = new int [ ] [5] [4] ;     //illegal
```

Delete operator is used to free memory space when there is no need of data object.
It is done as following:

delete pointer_variable

where pointer variable is that pointer which points to the data object created by new operator.

example

```
delete p ;
delete q ;
```

If we want to free memory space of array that has been allocated using dynamic method, following is done.

delete [size] pointer variable ;

2.15 I.O. Stream

We use #include directive in our program as given below :
#include<iostream>

This directive preprocessor tells to insert the contents of iostream file. It includes declaration of cout<< and "cin" >> operators. Header file should be written at the beginning of program in which input/output statements are being used.

2.16 Comparison of cout & cin with printf & scanf

We know C++ is the super set of C language, anything we do in C, and things we use in C, we can use them in C++ as well.

printf and scanf

printf( ) is used as an output function in C. scanf( ) is an input function. Both of these functions are defined in header file<stdio.h>. We can use these functions in C++ also.

In C++, two more functions are used for this purpose cout and cin.
Both of these functions are used with >> input and << output operator.

Note : If we talk about speed then printf( ) and scanf( ) work more faster than cout and cin. But printf( ) and scanf( ) become complex at the time of formatting whereas cout and cin does not face such problem while doing formatting.

Points to Remember

1. C++ is an object oriented programming.(OOP)
2. >> input operator is used for cin in C++.
3. << output operator is used for cout in C++.
4. It is necessary to declare variable in C++.
5. All operations of C are available in C++.
6. A function is a set of given instructions in a program which is created to perform a specific task.
7. Functions are similar to #define macros.
8. A variable is called a local variable when declared inside the function block.
9. A variable is called a Global variable when declared outside the function block.
10. New and delete operator are used for memory allocation and deallocation.

Exercise

1. Fill in the blanks

1. ..... operator is called output operator.
2. Inline functions are similar to ..... function.
3 C++ Functions of has ability to define ..... for arguments.
4. New operator is used for .....
5. C and C++ compiler supports all .....data type.

2. Answer the questions

1. Explain history of C++ language.
2. How the variable declaration is done in C++ ?
3. Write about operators used in C++.
4. Write a note on inline function.
5. What is function overloading ?
6. Write difference between Reference and Pointer.
7. Explain the process of memory allocation using new operator.
8. Write difference between cout and cin.

Lab Activity

Program 1:

```
// my first program in C++

#include <iostream.h>
using namespace std;

int main()
{
cout<< "Hello World!";
return 0;
}
```

Program2:

```
// my second program in C++

#include <iostream.h>

using namespace std;

int main()
{
cout<<"Hello World! ";
cout<<"I'm a C++ program";
return 0;
}
```

Program 3:

```
/* my second program in C++

with more comments */

#include <iostream.h>
using namespace std;

int main()
{
cout<<"Hello World! ";     // prints Hello World!
cout<<"I'm a C++ program"; // prints I'm a C++ program
return 0;
}
```

Program 4:

```
// operating with variables

#include <iostream.h>
```

```
using namespace std;

int main()
{
// declaring variables:
int a, b;
int result;

// process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

// print out the result:
cout<< result;

// terminate the program:
return 0;
}
```

Program 5:

```
// initialization of variables

#include <iostream.h>
using namespace std;

int main()
{
int a=5;               // initial value = 5
int b(2);              // initial value = 2
int result;            // initial value undetermined

  a = a + 3;
result = a - b;
cout<< result;

return 0;
}
```

Program 6:

```
// defined constants: calculate circumference

#include <iostream.h>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main()
{
double r=5.0;               // radius
double circle;

 circle = 2 * PI * r;
cout<< circle;
cout<< NEWLINE;

return 0;
}
```

Program 7:

```
// assignment operator

#include <iostream.h>
using namespace std;

int main()
{
int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
```

```
  b = 7;             // a:4,  b:7

cout<<"a:";
cout<< a;
cout<<" b:";
cout<< b;

return 0;
}
```

Program 8:

```
// compound assignment operators

#include <iostream.h>
using namespace std;

int main()
{
int a, b=3;
  a = b;
a+=2;              // equivalent to a=a+2
cout<< a;
return 0;
}
```

Program 9:

```
// conditional operator

#include <iostream.h>
using namespace std;

int main()
{
inta,b,c;

  a=2;
  b=7;
  c = (a>b) ?a : b;

cout<< c;

return 0;
}
```

Program 10:

```
/ i/o example

#include <iostream.h>
using namespace std;

int main()
{
int i;
cout<<"Please enter an integer value: ";
cin>> i;
cout<<"The value you entered is "<< i;
cout<<" and its double is "<< i*2 <<".\n";
return 0;
}
```

Program 11:

```
// function example
#include <iostream.h>
using namespace std;
int addition (int a, int b)
{
int r;
  r=a+b;
return (r);
}

int main()
{
int z;
```

```
  z = addition (5,3);
cout<<"The result is "<< z;
return 0;
}
```

Program 12

// function example

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

int subtraction (int a, int b)
{
int r;
  r=a-b;
return (r);
}

int main()
{
int x=5, y=3, z;
  z = subtraction (7,2);
cout<<"The first result is "<< z <<'\n';
cout<<"The second result is "<< subtraction (7,2) <<'\n';
cout<<"The third result is "<< subtraction (x,y) <<'\n';
  z= 4 + subtraction (x,y);
cout<<"The fourth result is "<< z <<'\n';
return 0;
}
```

Program 13

```
// void function example
#include <iostream.h>
using namespace std;

void printmessage()
{
cout<<"I'm a function!";
}

int main()
{
printmessage();
return 0;
}
```

Program 14

```
// passing parameters by reference
#include <iostream.h>
using namespace std;

void duplicate(int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}


int main()
{
int x=1, y=3, z=7;
  duplicate (x, y, z);
cout<< "x=" << x << ", y=" << y << ", z=" << z;
return 0;
}
```

Program 15:

```
// more than one returning value
```

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

voidprevnext(int x, int&prev, int& next)
{
prev = x-1;
next = x+1;
}

int main()
{
int x=100, y, z;
prevnext (x, y, z);
cout<<"Previous="<< y <<", Next="<< z;
return 0;
}
```

Program 16

```
// default values in functions
#include <iostream.h>
using namespace std;

int divide(int a, int b=2)
{
int r;
  r=a/b;
return (r);
}

int main()
{
cout<< divide (12);
cout<<endl;
cout<< divide (20,4);
return 0;
}
```

Program 17

```
// overloaded function
#include <iostream.h>
using namespace std;

int operate(int a, int b)
{
return (a*b);
}

float operate(float a, float b)
{
return (a/b);
}

int main()
{
int x=5,y=2;
float n=5.0,m=2.0;
cout<< operate (x,y);
cout<<"\n";
cout<< operate (n,m);
cout<<"\n";
return 0;
}
```

Program 18

```
// factorial calculator
#include <iostream.h>
using namespace std;

long factorial(long a)
{
if (a > 1)
return (a * factorial (a-1));
```

```
else
return(1);
}

int main()
{
long number;
cout<<"Please type a number: ";
cin>> number;
cout<< number <<"! = "<< factorial (number);
return 0;
}
```

Program 19

```
// declaring functions prototypes
#include <iostream.h>
using namespace std;

void odd(int a);
void even(int a);

int main()
{
int i;
do {
cout<<"Type a number (0 to exit): ";
cin>> i;
odd (i);
  } while (i!=0);
return 0;
}

void odd(int a)
{
if ((a%2)!=0) cout<<"Number is odd.\n";
else even(a);
}

void even(int a)
{
if ((a%2)==0) cout<<"Number is even.\n";
else odd(a);
}
```

Program 20

```
/ rememb-o-matic using new and delete operator
#include <iostream.h>
#include <new>
using namespace std;

int main()
{
int i,n;
int * p;
cout<<"How many numbers would you like to type? ";
cin>> i;
  p= new (nothrow) int[i];
if (p == 0)
cout<<"Error: memory could not be allocated";
else
  {
for (n=0; n<i; n++)
    {
cout<<"Enter number: ";
cin>> p[n];
    }
cout<<"You have entered: ";
for (n=0; n<i; n++)
cout<< p[n] <<", ";
delete[] p;
  }
return 0;
}
```

Exercise

1. Point out the errors, if any, in the following programs

   a.
```
void main()
{
int  a=30;
f();
}
void f()
{
int b=20;
}
```

   b.
```
#include <iostream.h>
void main()
{
void fun1(void);
void fun2(void);
fun1();
}
void fun1 (void)
{
fun2();
cout<<endl<<"hi…Hello";
}
void fun2(void)
{
cout<<endl<<"to you";
}
```

2. Show the output of the following Programs

   a.
```
#include <iostream.h>
void main()
{
floatr,a;
const float PI=3.14;
cin>>r;
a=PI*r*r;
cout<<endl<<"area of circle="<<a;
}
```

   b.
```
#include <iostream.h>
inline float mul(float x, float y)
{
        return(x*y);
}
inline double div(double p, double q)
{
        return (p/q);
```

```
}
int main()
{
float a=12.345;
float b=9.82;
cout<<mul(a,b) <<“\n”;
cout<<div(a,b) <<“\n”;
return 0;
}
```

Lesson - 3

C++ Class Concept

3.1 C++ Class Concept

Class is a procedure by which we join the related functions all together. It allows hide of Data and functions from external use if necessary. A new abstract data type to created which is just like any other built in data type. When class is defined.

```
class class name
{
    private:
variable declarations ;
    function declarations ;
public ;
variable declarations ;
    function declarations;
}
```

Body of class is written in parenthesis (braces) and it is ended with semicolon. Variable and functions are declared inside class body. These variables and functions are called class members.

These are divided into 2 parts - private and public from which it is specified that which member is private and which one is public. Private and Public keywords are called access specifiers.

The class members declared private, are accessed within class. Whereas public member can be accessed from outside also.

3.1. Data members and member functions : Variables which are declared in the class are called data members and functions are called member functions. Only member functions can access private data members and private functions.

3.1.2 Create an object : Once class is declared then we can create variables of that type by using class name.

for example

item *x* ; // memory for *x* is created

Class variables are known as object in C++. Therefore x is called one type (item) of object. We can declare more than one object in one statement.

for example:

item *x, y, z* ;

When class is defined after closing the bracket} an object can also be created by writing the object name.

```
class item
{
    .....
    .....
    .....
}
x, y, z ;
```

Accessing class members :

Format to call member functions is given below.
object_name : function function_name(actual arguments);

3.1.3 Defining member function :
Member functions can be defined by two ways:
1. Outside class definition
2. Inside class definition

3.1.3.1 Outside class definition :

syntax

```
return type class-name :: function-name(argument declaration)
{
    function body
 }
```

Label class name : : It tells compiler that the given function is related to class which means it defines the scope of function. The character: : is called scope resolution operator.

```
void item :: get data (int a, float b)
{
    number = a ;
    cost = b ;
}
    void item :: putdata (void)
{
    cout << ‘‘number :’’
    cout << ‘‘cost :’’ << cost << ‘‘\n’’ ;
```

3.1.3.2 Inside the class Definition

When function is defined inside the class, it behaves like inline function. To define inside the class is as follows :

example :

```
class item
{
    int number ;
    float cost ;
    public :
void getdata (int a, float b) ; //declaration
// inline function
void putdata (void) ; //definition inside the class
 {
    cout << number << ‘‘/n’’ ;
    cout << cost << ‘‘/n’’ ;
 }
};
```

A function is said to be inline function, when it is defined inside the class.

3.2 Static data members

Data members of class can be static. A static member has some special features as given below.

* When first object of class is created then static member is initialized to 0. There is no permission for other initialization.
* Only one copy is created for static member of whole class and it is shared by all objects of the class.
* It works all the time in program but is seen in class only.

Static variables are used for maintenance of those values which are common to all classes.

3.3 Static member function

Static functions are like static variables. A member functions which is made static has features as given below.

* Static functions can access those static members which are declared in same class.
* A static member function can be called by using single class as shown below :

class_name : : function name;

Program given below implement these characteristics

static member functions (program)

```
# include < iostream >
using namespace std ;
class test
{
    int code ;
    static int count ;  //static member variable
    public :
    void setcode (void)
{
    code = ++ count ;
}
void showcode (void)
{
    cout << ‘‘object number :’’ << code << ‘‘/4’’ ;
}
    static void showcount (void) // Static member function
  {
    cout << ‘‘count :’’ << count << ‘‘\n’’ ;
  }
};


int test :: count ;
int main ( )
{
test t1, t2 ;
t1 . setcode ( ) ;
t2 . setcode ( ) ;
test :: showcount ( ) ; // accessing static function
test t3 ;
t3 . setcode ( ) ;
test :: show count ( ) ;
t1 . show code ( ) ;
t2 . show code ( ) ;
t3 . show code ( ) ;
return 0 ;}
```

3.4 Friendly Functions

We keep saying in this exercise from the beginning that private member can not be accessed from outside the class.

A non member function cannot access the private data of class. In that case we allow to share one function by two classes. In this case, common function in C++ is made friendly to both classes and functions are allowed to access the private data of classes. That function need not be member of any class.

We have to directly declare a function as a friend to make outer function friendly to a class.

example :

```
class ABC
{
        .................
        .................
        public
        .................
        .................
        friend void xyz (void) ; //declaration
};
```

function declaration is done with keyword friend. It can be declared anywhere like any other variable.

Friend Function

```
# include < iostream >
using namespace std ;
class Sample
{
    int a ;
    int b ;
public :
    void setvalue ( ) {a = 25 ; b = 40 ;}
    friend float mean (sample S) ;
};
    float mean (sample S)
{
    return float (S.a + S.b) / 2.0 ;
}
    int main ( )
  {
    sample x ;    // object X
    X . setvalue ( ) ;
    cout << ‘‘Mean Value = ‘‘ << mean (x) << ‘‘/n’’ ;
    return 0 ;
}
```

3.5 const. Member Functions

If any member function does not want to alter data of a class, then it is built as const member function.

It can be declared as below :

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

3.5.1 Constant class objects

Like any other built in data type we can make constants to objects of class.

```
const int nvalue = 5;
const int nvalue 2(7)
```

In case of classes, initialization can be done through constructors.

```
const Date CDate;
const Date CDate2(10,16,20,20);
```

When a class object is initialized through constructor then any effort made to modify member variable does not succeed.

example:

```
class Something
{
public :
    int m_nValue;
    Something ( ) { m_nValue = 0 ; )

    void Resetvalue ( )  { m_n Value = 0 ; }
    void Setvalue (int Value)  { m_n Value = nValue; }

    int Getvalue ( )  {return m_n Value; }
} ;
int main ( )
{
    const Something cSomething; // calls default constructor

    cSomething.m_nValue = 5 ; //violates const
    cSomething.ResetValue( ) ; //violates const
    cSomething.SetValue (5) ; //vilolates const
    return 0 ;
}
```

3.6 Pointers to Members

Address of class member can be assigned to a pointer and that address can be taken from fully qualified operator. A class member can be declared by using : : operator.

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

We can defined member with m pointer as shown below

```
Int A : : *iP = &A : : m;
```

In above written line (A : : *) means (Pointer_ to member of A class) and (&A: :m) means address of the m member of A class.

3.7 Constructors

Constructor is an important/member function which is used to special initialize the object of class. It is special since its name is like class name. Constructors are called when objects are created related to it. They are called constructors because they construct the values of class data members. Constructors are declared and defined n following way.

```
// class with constructor
class integer
{
    int m, n ;
    public ;
    integer (void) ;   //constructor declare
    .....
    .....
},
integer :: integer (void)  //constructor defined
  {
    m = 0 ; n = 0 ;
}
```

Characteristics of constructors functions are given below:-

* They should always be declared in public section.
* They are called automatically when objects are created.
* They cannot be taken from derived class and can call base class constructor.
* Constructors cannot be virtual.
* We cannot call their addresses.
* An object which is with constructor, it cannot be used as member of union.

example :

class with constructor

```
# include < iostream >
using namespace std ;
class integer
  {
    int m, n ;
    public :
    integer (int, int) ;                  //constructor declared
    void display (void)
    {
    cout << ‘‘m = ’’ << m << ‘‘\n’’ ;
    cout << ‘‘n = ’’ << n << ‘‘\n’’ ;
  }
}
    integer :: integer (int x, int y)
    {
    m = x ; n = y ;                       //constructor defined
    }
    int main( )
    {
    integer int1 (0, 100) ;               //constructor called implicit g.
    integer int2 = integer (25, 75) ;     //constructor called explicitly
    cout << ‘‘\n OBJECT 1’’ << ‘‘\n’’ ;
    int 1. display ( ) ;
    cout << ‘‘\’’ OBJECT 2’’ << ‘‘\n’’ ;
    int2 display ( ) ;
    return 0 ;
}
```

3.8 Overloading constructor:

When more than one constructor function are defined in a class, it is called overloading constructor.

Program :

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


class complex
{
    float x, y ;
public :
    complex( ) & }
    complex (float a) {x = y = a; }
    complex (float real, float imag)
    {x = real ; y = imag ; }
friend complex sum (complex, complex) ;
    friend void show (complex) ;
};
complex sum (complex C1, complex C2)
{
    complex C3 ;
    C3.x = C1.x + C2.x ;
    C3.y = C1.y + C2.y ;
    return (C3) ;
}
void show (complex C)
{
   cout << C.x << ‘‘+ j’’ << C.y << ‘‘\n’’ ;
}
int main ( )
{
    complex A (2.7, 3.5) ;
    complex B (1, 6) ;
    complex C ;
    C = Sum (A, B) ;
    cout << ‘‘A = ’’; Show (A) ;
    cout << ‘‘B = ’’ ; Show (B) ;
    cout << ‘‘C = ’’ ; Show (C) ;
    return 0;
}
```

3.9 Copy (Constructor)

A copy constructor is used to declare and initialize one object from another object.

example : integer I2(II)

Where I2 defines an object and at the same time values of I2 becomes the values of III. The process of initialization by copy construct or is called copy initialization.

3.10 Destructor:

As it is clear from the name itself, the objects which are created by constructors are deleted by destructors.

A destructor is also member function like constructors which has same name as of class name but its name is written with prefix character (~) Tilde.

A destructor neither takes argue nor return value.

When we come out of program, compiler calls it automatically so that this function clears the memory space. Which is not in use.

3.11 C++ this Pointer

C++ uses a special pointer (this pointer) to access address of every object. This pointer is an implicit parameter for all member function.

Friend function does not have this pointer.

```
# include <iostream.h>
using namespace std;
class Box
{
    public :                    // Constructor definition

        Box (double 1 = 2.0, double b = 2.0, double h = 2.0)
        {
            cout <<‘‘Constructor called.‘‘ << end1;
            length = l ;
            breadth = b ;
            height = h ;
        }
        double Volume ( )
        {
            return length * breadth * height ;
        }
        int compare (Box box)
        {
            return this->Volume ( ) > box.Volume ( ) ;
        }
    private :
        double length;              // Length of a box
        double breadth;             // Breadth of a box
        double height;              // Height of a box
 };
int main (void)
{
    Box Box1 (3.3, 1.2, 1.5) ;      // Declare box1
    Box Box2 (8.5, 6.0, 2.0) ;      // Declare box2
    if (Box1.compare (Box2))
    {
        cout << ‘‘Box2 is smaller than Box1’’ <<end1;
    }
    else
    {
        cout << ‘‘Box2 is equal to or larger than Box1’’ <<end1;
    }
    return 0;
}
```

3.12 Empty classes

We can declare empty classes but size of these class objects is non-zero.

Example :

```
// empty_classes.cpp
// compile with : /EHsC
# include <iostream.h>
class NoMembers
{
};
using namespace std ;
int main ( )
{
    NoMembers n;  // Object of type Nomembers.
```

```
    cout << ‘‘The size of an object of empty class is :’’
         << size of n << endl;
}
```

Output

The size of an object of empty class is : 1.

Memory is among these non-zero size objects therefore these objects allocated have unique memory address.

3.13 Assignment vs. Initialization

Assignment and initialization both are different operations and have different uses. Assignment and initialization differ because they are used in different context and do different tasks. This difference does not implemented on built in data-type for e.g. int or double because both of these operations copy only few bits in this case.

```
int a = 12;        //initialization, copy 0 × 000c to a
a = 12 ;           //assignment, copy 0 × 000c to a
```

It is done in different way in case of user defined data type as explained in example given below :-

```
class string {
    public :
    string (const char * init) ;  //intentionally not explicit !
    ~ string ( ) ;
    string (const string & that),
    string & operator = (const. string & that) ;
    string & operator = (const. char * Str) ;
    void Swap (string & that) ;
    friend const string // concate ~ at r
    operator + (const string & const string &) ;
    friend bool operator < (const string &, const string &) ;
    // .....
    private
    string (const char *, const char *) ;  // Computational
    char * & ;
 };
```

String Object initialized the character string in following way.

```
string :: string (const char * int) &
if (!int) init = ‘‘  ’’;
S_ = new char [strlen (init) + 1];
strcpy (S_1 init) ;
}
```

Destructor will do the work it knows.

```
string :: ~ string ( ) & delete [ ]  S_, }
```

Assignment is a tough task than construction.

```
Sstring & string :: operator = (const char * str)}
if (! str) str = ‘‘ ’’ ;
char * imp = strcpy [new char [strlen (str) + 1], str) ;
delete[ ] S_,
S_ = temp;
return * this ;
}
```

3.14 class vs. Object

Object : There are two main characteristics of real world object - state and behavior. for example state human being(name, age) and behavior (running, sleeping). Software objects are also resembles real world objects. Which have its own field and functions. which has its own field and functions.

Class : Class is a type of template or blue print which is used to create an object. A class consists of field, static field, method, static method and constructors . Field holds the state of class and Method holds behavior of class.

In brief it can be said that an object is software bundle of related state and behavior and class is that blue print which is used to create an object.

3.15 Array of Objects

Array of those variables whose type is class is called array of objects. The identifier used to refer array of objects, is a data type defined by user.

Example :

```
# include <iostream.h>
const int MAX = 100 ;
class Details
 {
   private :
        int salary ;
        float roll;
   public :
        void getname ( )
            {
                cout << ‘‘\n Enter the Salary :’’ ;
                cin >> salary ;
                cout << ‘‘\n Enter the roll :’’ ;
                cin >> roll ;
            }
        void putname ( )
            {
                cout << ‘‘Employees’’ << salary <<
                ‘‘and roll is’’ << roll << ‘\n’ ;
            }
        };

   void main ( )
        {
            Details det [MAX];
            int n = 0;
            char ans ;
            do {
                cout << ‘‘Enter the Employee Number ::’’ << n + 1;
                det [n++].getname;
                cout << ‘‘Enter another (y/n) ? :’’ ;
                cin >> ans ;
            } while (ans ! = ‘n’) ;
            for (int j = 0 ; j < n ; j++)
                {
                    cout << ‘‘\nEmployee Number is ::’’ << j + 1;
                    det [j].putname ( ) ;
                }
        }
```

3.16 Overloading of New and Delete operator

We can overload new & delete operator like any other C++ operator. But during this process we should be careful while accepting parameter, value to return and place to declare.

We know new operator in C++ is used for memory allocation. Malloc ( ) function is used for the same in C.

What is the need to overload new & delete operator?

1. To control memory allocation.
2. To keep track of memory allocation and deal location.
3. To perform other operations during memory allocation.

The example given below explain very easily about overloading of new & delete operator.

```
void * operator new (size_t new)
{
    return malloc (nuon);
}
void operator delete (void * ptr)
{
    free (ptr) ;
}
```

Example

The following example overloads the + operator to add two complex numbers and returns the result.

```
// operator_overloading.cpp
// compile with : /EHsc
# include <iostream.h>
using namespace std;
class complex
{
public :
    complex (double r, double i) : re(r), im(i) { }
    complex operator + (complex & other) ;
    void Display { } {cout << re << ‘‘,’’ << im << endl;}
private :
    double re, im;
};
// operator overloaded using a member function
complex complex :: operator + (complex & other)
{
return complex (re + other.re, im + other.im);
}
int main ( )
{
    complex a = complex (1.2, 3.4) ;
    complex b = complex (5.6, 7.8) ;
    complex c = complex (0.0, 0.0) ;
    c = a + b ;
    c. Display ( );
```

A simple Message Header

```
// sample of operator Overloading
# include <string>
class PlMessageHeader
{
    std :: string m_ThreadSender;
    std :: string m_ThreadReceiver;
    //return true if the messages are equal, false otherwise
    inline bool operator == (const PlMessageHeader & b) const
    {
        return ((b.m_Threadsender == m_ThreadSender) &&
            (b.m_ThreadReceiver == m_ThreadReceiver));
    }
    //return true if the message is for name
    inline bool is For (const std :: string & name) const
    {
        return (m_ThreadReceiver == name) ;
    }
    //return true if the message is for name
    inline bool is For (const char * name) const
    {
        return (m_ThreadReceiver == name); //since name type is std :: string, it becomes unsafe if name ==
```

```
        NULL
    }
};
```

3.17 Type Conversion

It is very common thing to show same type of data with different ways by means of scientific calculations. This includes conversion of data from one form to another for example conversion of radian to degree, polar to rectangular etc. In C++, this conversion process is done by overloading assignment operator.

Example

```
var1 =var2
```

where var1 and var2 both are of integer type. Similarly values of operators created in a class can be assigned to each other.

For example

C3= C1 + C2//C1,C2 and C3 are the objects of same class.

Same type of data items are converted by compiler and no effort is done by user. If the data item are different. Then user have to create function for data conversion.

3.17.1 Conversion between Basic data types

example weight = age

where data type of weight is float and data type of age is integer.

Here compiler calls a special function which convert the value of age into float. Compiler has many such built in function which converts basic data types like char to int, float to double etc.

The quality of compiler in which conversion is done without help of user, is called implicit type conversion. Instructions are given to compiler from outside or type conversion using type cast operator for example instruction to convert int to float as

```
weight = (float) age;
```

This word float which enclosed in braces is called type cast operator. Only those function will accept explicit conversion from float to int which are accepted by implicit conversion.

3.17.2 Conversion between Object and Basic Data Types

Compiler can convert those built in data types which are supported by programming language. We cannot rely on compiler for conversion of primitive data types of user-defined data types because compiler does not know of logical meaning of user defined data types. Therefore for a meaning full conversion, conversion function has to be built as required. The process of conversion between user defined data types and basic data types are explained through program meterapp.

Where and How the Conversion function should exist ?

For conversion of basic types to user defined type conversion function, like a constructor, should be defined in objects of user defined class.

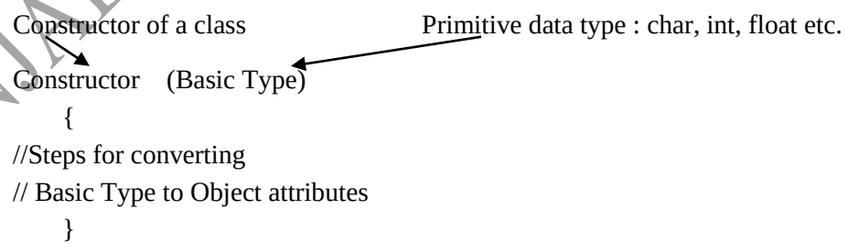


Fig. Conversion function : basic to user-defined

Similarly conversion of Basic data type from the user defined type conversion function, like operator function, should be defined in objects of user defined class.

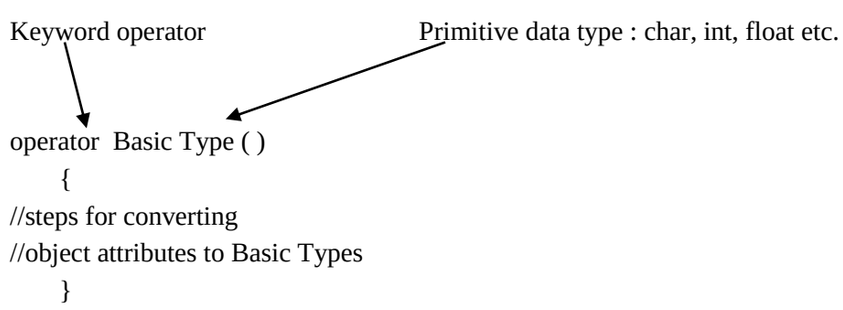


Fig. Conversion Function : User defined to Basic

Example : **me**ter.cpp

```
//Conversion from Meter to Centimeter and Vice-versa
# include < iostream.h >
//Meter class for MKS measurement system
class Meter
{
    private :
    float length       //length in meter
    public :
    Meter ( )       //constructor O, no arguments
    {
        length = 0.0 ;
    }

//Conversion for Basic data-item to user-defined Type
//Initlength is in centimeter Unit
Meter (float Initlength) //Constructorl, one argument
{
    Length = Initlength/100.0 ;   //Centimeter to meter
}
//Conversion from user-defined type to Basic data-item
//i.e. from meter to centimeter
    operator float ( )
    cout << * length (in meter = '' << length ;
  }
};
void main ( )
{
    //Basic to user-defined conversion demonstration section
    Meter meter1;   //user constructor 0
    float length1;
    cout << ''Enter length (in cms): '' ;
    cin >> length1 ;
    meter1 = length1; //converts basic to user-defined uses construction
    meter1 . show length ( ) ;
    //user defined to Basic conversion demonstration section
    Meter meter 2 ; //uses constructor 0
    float length2;
        meter2. (Tet length( ) ;
    Length2 = meter2 ;   //converts user = defined to basic uses operator float)
    cout << * Length (in cms) = '' << Length 2'' ;
    }
```

Output :

```
Enter length (in cms.) : 1500
Length (in meter) = 1.5
Enter length (in meters) : 1.669
Length (in cms.) = 166.900009
```

Things to remember

1. Class is a procedure in which data and its related functions are collectively joined.
2. In a class, data member and Member functions can be static.
3. Constructor is a special member function which is used to initialize the object of class.
4. A copy constructor is used to initialize from one object to another object.
5. Array of variables is called array of objects which has its type "class".
6. Conversion of one type to another type of data is called type conversion.

Exercise

1. Fill in the blanks

1. ..... are always declared in public section.

2. The properties of objects are ..... and .....

3. The process of initialization through copy constructors is called .....

4. Member functions are defined by ..... types.

5. When a function is defined in a class is known as .....

2. Answer the questions

1. Write a brief note on scope resolution operator (: :)?
2. Write a note on static data member and member functions.
3. Explain class concept?
4. What is constructor. Give example. How does it differ from destructor.
5. Write the process of conversion between objects and basic data types.
6. Differentiate object and class?
7. What is array of objects. explain with example.
8. What is the need to overload new and delete operator?
9. What are friend functions?
10. Why does "this" pointer used in C++?

Lab Activity

Classes

Program 1:

```
// classes example
#include <iostream.h>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Program 2:

```
// example: one class, two objects
#include <iostream.h>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Program 3:

```
// example: class constructor
#include <iostream.h>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area () {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Program 4:

```
// example on constructors and destructors
#include <iostream.h>
using namespace std;

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
```

```
}

int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Program 5:

```
// overloading class constructors
#include <iostream.h>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Program 6:

```
// pointer to classes example
#include <iostream.h>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
};


void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() << endl;
  cout << "d[1] area: " << d[1].area() << endl;
  delete[] d;
  delete b;
  return 0;
```

```
}
```

Classes -II

Program 7:

```
// vectors: overloading operators example
#include <iostream.h>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

Program 8:

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

class CDummy {
  public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b";
  return 0;
}
```

Program 9:

```
// static members in classes
#include <iostream.h>
using namespace std;

class CDummy {
  public:
```

```
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

Pointers

Program 10:

```
// my first pointer
#include <iostream.h>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

Program 11:

```
// more pointers
#include <iostream.h>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;

  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

Program 12:

```
// more pointers
#include <iostream.h>
using namespace std;

int main ()
{
  int numbers[5];
```

```
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

Program 13:

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;
}
```

Program 14:

```
// pointer to functions
#include <iostream.h>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);                                    8
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

Destructor

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

class myclass {
  int *p;
```

```
public:
  myclass(int i);
  ~myclass();
  int getval() { return *p; }
};

myclass::myclass(int i)
{
  cout << "Allocating p\n";
  p = new int;
  if(!p) {
    cout << "Allocation failure.\n";
    exit(1); // exit program if out of memory
  }

  *p = i;
}

// use destructor to free memory
myclass::~myclass()
{
  cout << "Freeing p\n";
  delete p;
}

void display(myclass &ob)
{
  cout << ob.getval() << '\n';
}
int main()
{
  myclass a(10);

  display(a);

  return 0;
}
```

Exercise

1. Find out the errors if any:

(a)

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

class cl {
  int i; // private by default
public:
  int get_i();
  void put_i( j);
};

int cl::get_i()
{
  return i;
}

void cl::put_i(j)
{
  i = j;
}

int main()
{
  cl s;

  s.put_i(10);
  cout << s.get_i() <<endl;

  return 0;
}
```

(b)

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

// Class to represent a box
class Box
```

```
{
  public:
    double length;
    double breadth;
    double height;

    // Inline initialization
    Box(double lv = 1.0, double bv = 1.0, double hv = 1.0):length(lv),
                                                           breadth(bv),
                                                           height(hv)
    {
      cout << "Box constructor called" << endl;
    }

    // Function to calculate the volume of a box
    double volume()
    {
      return length * breadth * height;
    }
};

int main()
{
  Box firstBox(80.0, 50.0, 40.0);

  // Calculate the volume of the box
  double firstBoxVolume = firstBox.volume()
  cout << endl;
  cout << "Size of first Box object is "
       << firstBox.length  << " by "
       << firstBox.breadth << " by "
        << firstBox.height
       << endl;
  cout << "Volume of first Box object is " << firstBoxVolume
  return 0;
}
```

Lesson -4
Inheritance

4.1 Inheritance

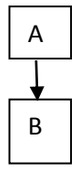
C++ supports the concept of Reusability or repeatedly use.
Classes of C++ can be reused in multiple ways. After writing and testing of class, it can be used by other programmers as required.

Basically in this way newly created class can inherit or reuse the properties of existing class. The process in which new class is prepared from an old class is called Inheritance. Old class is called base class and new class is called derived/sub class.

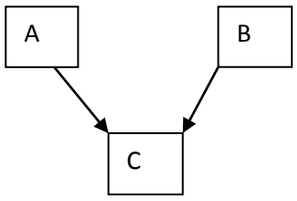
4.1.1 What is a derived class?

Derived class inherits or gets some or all properties from base class. Sometimes in a class more than one properties passed. A derived class which has only one base class is called single inheritance and more than one base classes is called multiple inheritance. Traits of one class can be inherit from more than one classes. This process is called hierarchical inheritance.

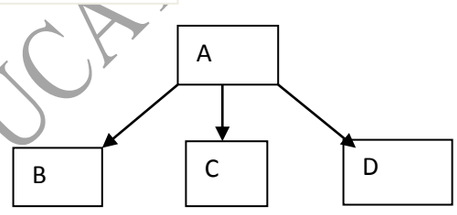
The procedure in which one class is derived from another derived class. This is called Multilevel Inheritance.



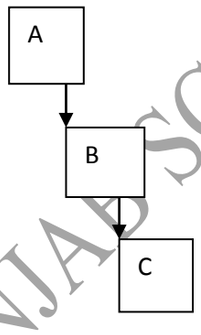
Single Inheritance



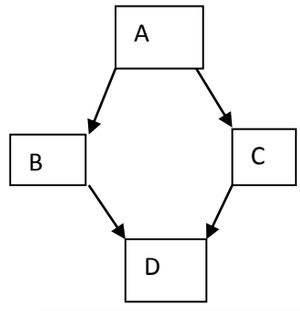
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

4.1.2 Defining derived class

A basic derived class is defined in following way.

```
class derived_class_name : : visibility_mode
{
```

```
            ...............//
            ...............// members of derived class
            ...............//
};
```

4.4 Visibility Mode or Access Specifier

It specifies that properties of base class are either of private derivation or public derivation.

1. class ABC : private XYZ
   {
   members of ABC // private derivation
   };
2. class ABC : public XYZ
   {
   members of ABC //public derivation
   };
3. class ABC : XYZ
   {
   members of ABC //private derivation by default
   };

4.2.1 private Derivation

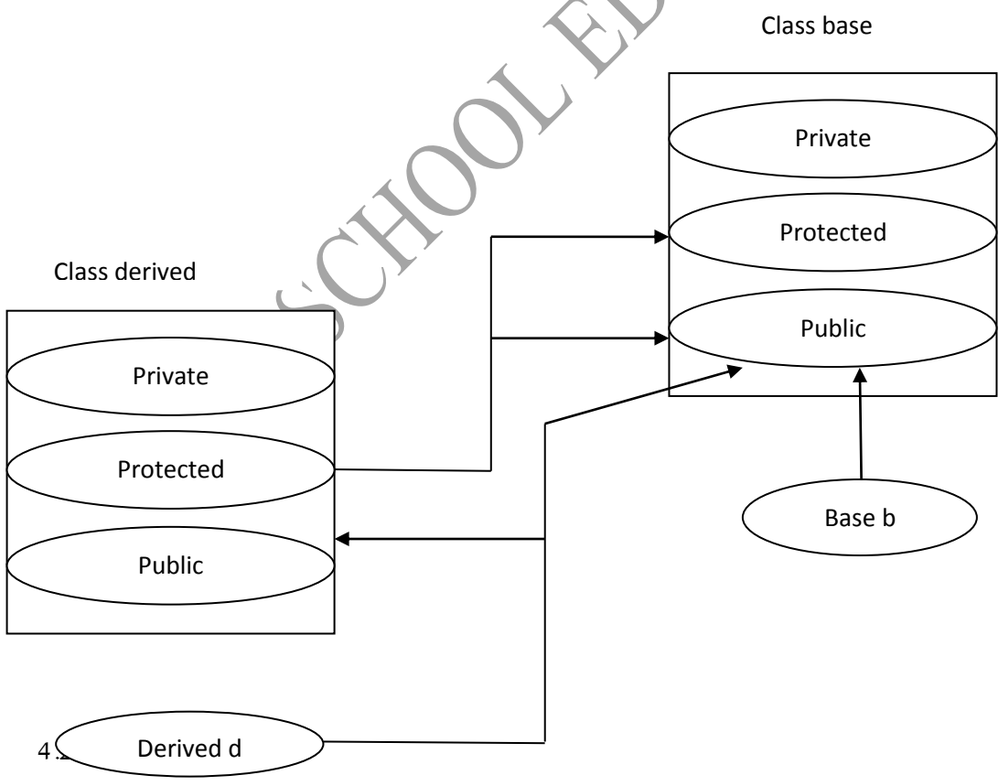
When a base class is inherited individually from derived class, then the public member of base class becomes private members of derived class. Therefore public members of base class can only be accessed by function members of derived class.

4.2.2 public Derivation

When a base class inherited as public then public members of base class becomes public members of derived class.

4.2.3 Protected Derivation

A protected member can be accessed by their own members on function of own class.



4.2.4 Virtual Derivation

Virtual derivation can be understood from following example.

```
Class A   // grand parent
class A        //grand parent
```

```
{
    .................
    .................
};
class B1 : virtual public A  //Parent 1
{
    .................
    .................
};
class B2 : public virtual B  //Parent 2
{
    .................
    .................
};
class C : public B1, public B2   //Child
{
    .................  //only one copy of A
    .................  //will be inherited
};
```

When a class is made a Virtual Base class, then it is assured that only one copy is inherited. Though there are more than one inheritance path exist between virtual base class and derived base class. Keywords virtual and public can be used in any series.

4.3 Function Overriding

As we have discussed earlier, overloading exists when there are two functions with same names but either having different argument type or number of arguments are different. Compiler has to generate code in every case. This work can be done both way inside or outside the class.

Overriding exists when you derive child class from base class. At the same time, methods or function of base class replaces or changes the child class function (which has same no. of arguments or of one type)

4.4 Overloading vs. Overriding

1. Overloading of function occurs that time when one class is inherited from other class. Overriding can occur without Inheritance.
2. Function signature should be separate in overloading. which means number of arguments or type of arguments should be different function signature should b same in overriding.
3. Overriding functions are separate in case of scope but overloading function exists in one scope.
4. When derived class has to do some more work or different work on base class, it requires overriding.
5. Overloading is used when we require function with same name but different arguments.

Overloading is the definition in which there are several functions with same name but having different arguments or different number of arguments.

example :

```
void Foo(int);
void Foo(int, int) ;
void Foo(char) ;
```

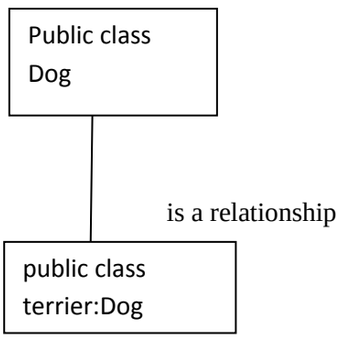
example :

```
class base
{
    public :
    int Foo ( )
    {
        return 1 ;
    }
};
class derived : public base
{
    public :
    int Foo ( )
    {
        return 2;
    }
```

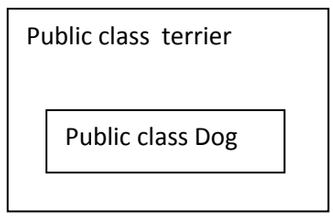
```
};
```

Overriding is a function which is defined in base class, its new definition is written in derived class.

4.5 **Inheritance v/s Containment**



Inheritance
(white-box)



containment
(block- box)

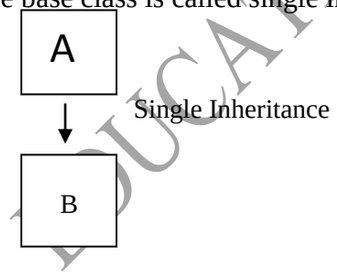
We can call Inheritance a white box which reuses the code. Which means derived class includes some internal qualities of base class. It tells the concept of Encapsulation.

There is another way to reuse the code in C++, which is called containment and it is called black box of reusability of code, Instead of deriving from base class, a new class creates an instance only of base class and new class accesses the public members of base through this instance. Instance of base class can be called Black box, because no access remains on implementation detail of derived class.

Note : We can create many(a number of) instances of objects inside a class.

4.6 **Single Inheritance**

A derived class which is derived from only one base class is called single inheritance.



Let us understand with an example. In the following program, there is a base class B and Derived class D. In Class B, there are one private data member one public data member and three member functions. In class D, there are one private data member and two public member functions

```
# include <iostream.h>
class B
{
    int a ;             //private ; note inheritable
    public :
    int b;              //public : ready for inheritance
    void get_ab ( ) ;
    int get_a (void) ;
    void show_a (void) ;
};
class D : public B    //public derivation
{
    int C ;
    public :
        void mul (void) ;
        void display (void) ;
};
void B :: get_ab (void)
{
    a = 5 ; b = 10;
}
int B :: get_a ( )
{
    return a;
}
```

```
void B :: show_a ( )
{
cout << ‘‘a = ’’ << a << ‘‘\n’’ ;
}
void D :: mul ( )
{
c = b* get_a ( )
}
void D :: display ( )
{
    cout << ‘‘a =’’ << get_a ( ) << ‘‘\n’’ ;
    cout << ‘‘b =’’ << b << ‘‘\n’’ ;
    cout << ‘‘c =’’ << c << ‘‘\n\n’’
}


//.....
int main ( )
{
    D d;
    d.get_ab ( ) ;
    d.mul ( ) ;
    d.show_a ( )
    d.display ( ) ;
    d.b = 20 ;
    d.mul ( );
    d.display ( );
    return 0 ;
```

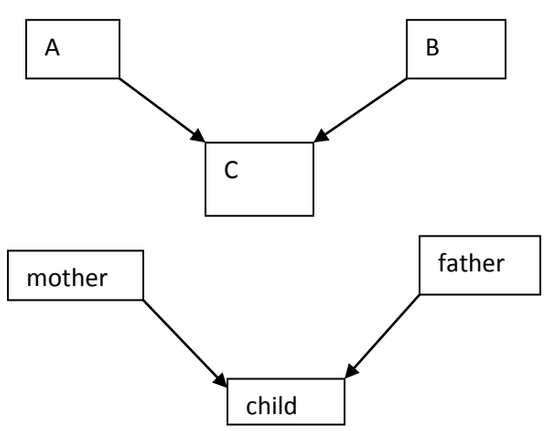
Output :

a = 5
a = 5
b = 10
c = 50
a = 5
b = 20
c = 100

Where class D is public derivation of class B. Therefore Class D inherits all the public member of Class B, therefore, public member of (base class) B class is also public member of class (derived class). Private members of B class cannot be inherited by D class.

4.7 **Multiple Inheritance**

A derived class which has more than one base class as is said to be multiple Inheritance.



One class, can inherit attributes of more than one class. Multiple inheritance allows built a new class, by combining features of more then one classes can be used as starting point. It is like a small child which inherits some features for his mother and some from his father.

Example:--

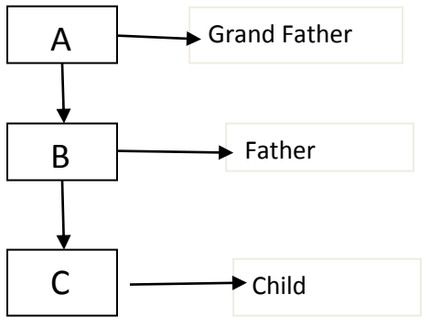
```
# include <iostream.h>
class M
{
    protected :
        int m;
    public :
    void get_on (int) ;
};
Class N
{
    protected
        int n ;
    public :
    void get_n (int);
};
Class P : public M, public N
{
    public :
        void display (void) ;
};
void M :: get_on (int X)
{
    m = x;
}
void N :: get_n (int y)
{
    n = y;
}
void P :: display (void)
{
    cout << ‘‘m = ’’ << m << ‘‘\n’’ ;
    cout << ‘‘n = ’’ << n << ‘‘\n’’ ;
    cout << ‘‘m * n=’’ << m * n << ‘‘\n’’ ;
}
int main ( )
{
    P . P ;
    P. get_m (10) ;
    P. get _n (20) ;
    P.display ( ) ;
    return 0 ;
}
```

Output :

```
m = 10
n = 20
m * n = 200
```

4.8 **Multilevel Inheritance:--**

This is a process in which a class is derived from a derived class, called multilevel inheritance.



Where class A is base class of derived class B is base class of derived class C. Class B is known an intermediate Base Class since it makes a link of inheritance between class A and Class C Chain ABC is called Inheritance path.

Example:-

```
class A : { ..... };    // Base class
class B : public A {.....} ; //B derived from A
class C : public B {.....} ;  //C derived from B
```

Example:-

```
#include <iostream.h>
class student
{
    protected :
        int roll_number ;
    public :
        void get_number (int) ;
        void put_number (void) ;
};

void student :: get_number (int a)
{
    roll_number = a;
}
void student :: put_number ( )
{
cout << ‘‘Roll Number : ’’ << roll_number << ‘‘\n’’ ;
}
class test : public student  //first level derivation
{
    protected :
        float sub1 ;
        float sub 2 ;
    public :
    void get_marks (float, float);
    void put_marks (void);
};
void test :: get_marks (float x, float y)
{
    sub1 = x ;
    sub2 = y ;
}
void test :: put_marks ( )
{
    cout << ‘‘Marks in SUB 1 = ’’ << sub1 << ‘‘\n’’ ;
    cout << ‘‘Marks in SUB 2 = ’’ << sub2 << ‘‘\n’’ ;
}
class result : public test  //second level derivation
```

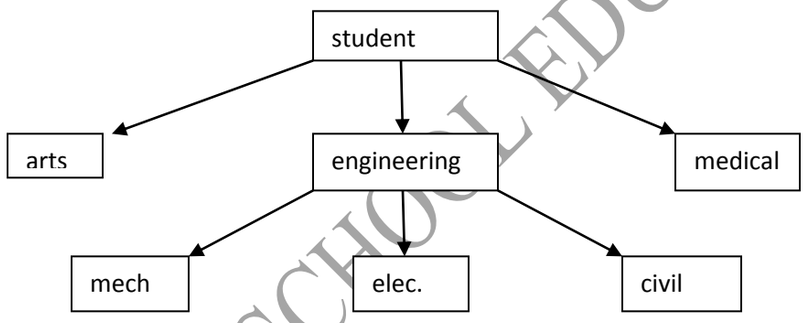
```
{
    float total ;   //private by default ;
    public
        void display (void) ;
};
void result :: display (void)
{
    total = sub1 + sub2 ;
    put_number ( ),
    put_marks ( );
    cout << ‘‘Total =’’ << total << ‘‘\n’’ ;
}
int main ( )
{
    result student 1 ;    // student 1 created
    student 1 . get_number (111) ;
    sutdent 2 . get_number (75.0, 59.5) ;
    student 1 . display ( ) ;
    return 0 ;
}
```

Output :

Roll Number : 111
Marks in SUB 1 = 75
Marks in SUB 2 = 59.5
TOTAL = 134.5

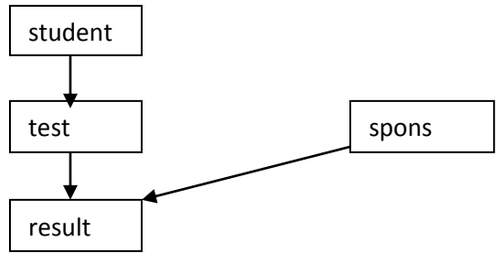
4.9 **Hierarchical Inheritance**

When attributes of one class are inherited by more than one class, this process is called hierarchical inheritance.



4.10 **Hybrid Inheritance**

If situation comes, where we have to design a program using more than one type of inheritance, newly constructed inheritance will be called hybrid inheritance.



4.11 **Conversion between Base class and Derived class**

As far as we talk about C++, A base class instance can not be converted to derived class without using a conversion operator. If you have Derived class instance stored as base class variable, then you can cast it in Derived Class.

```
class Base {
    public :
    Base { }  { }
    virtual void f ( ) {
    cout << ‘‘f base’’ << end l ;
```

```
    {
};
class Derived : public Base {
public :
    Derived ( ) { }
    void f ( ) {
    cout << ‘‘f derived’’ << end l;
    }
};
int main( )
{
    Base * b = new Derived ( ) ;
    Derived * d = (Derived *) b ;
    Base * b2 = new Base ( );
    Derived * d2 = (Derived *) b2;
    b → f( ) ;      // f derived
    b2 → f ( ) ;   // f base
    d → f ( ) ;    // f derived
    d2 → f ( ) ;  // base
    return 0 ;
}
```

Points to Remember :

1. The Process in which new class is created from old class is called Inheritance.
2. Visibility mode or access specifier specifies that features of base class are derived on private basis or derived on public basis.
3. Overriding consists of a function which is defined in a base class, it new definition is written in derived class.
4. Inheritance has many types for example single, multiple, multilevel, Hierarchical, and Hybrid Inheritance.

Exercise

1. ***Fill in the blank***

   1. After writing a class , It can be …….
   2. Old class is called ……. And new class is called……
   3. A derived class which has only one base class is known as ……
   4. ……..…... of function occurs when one class is inherited from another class.
   5. Reusability of …….. Code is said to be black.

2. ***Answer the Questions***

   1. Explain the process of inheritance.
   2. Write types of Inheritance.
   3. Write a note on access specifier.
   4. What is the meaning of function overloading.
   5. Differentiate between multiple and multilevel inheritance.
   6. Explain single Inheritance with example.

LAB ACTIVITY

1.

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

class A {
int data;
public:
void f(intarg) { data = arg; }
int g() { return data; }
};
class B : public A { };

int main() {
   B obj;
obj.f(20);
cout<<obj.g() <<endl;
}
```

2.

```
#include<iostream.h>
usingnamespacestd;
class A
{
public:
void display()
{
cout<<"Base class content.";
}
};

class B :public A
{

};

class C :public B
{

};
```

```
int main()
{
    C c;
c.display();
return0;
}
```

3.

```
#include <iostream.h>

using namespace std;

// Base class
class Shape
{
public:
voidsetWidth(int w)
      {
width = w;
      }
voidsetHeight(int h)
      {
height = h;
      }
protected:
int width;
int height;
};

// Derived class
class Rectangle: public Shape
{
public:
intgetArea()
      {
return (width * height);
      }
};
int main(void)
{
   Rectangle Rect;

Rect.setWidth(5);
Rect.setHeight(7);

   // Print the area of the object.
cout<< "Total area: " <<Rect.getArea() <<endl;

return 0;
}
```

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

```
Total area: 35
```

4.

```
#include<iostream.h>

usingnamespacestd;

// Base class Shape
classShape
{
public:
voidsetWidth(int w)
{
width= w;
}
voidsetHeight(int h)
{
height= h;
}
protected:
int width;
```

```
int height;
};

// Base class PaintCost
classPaintCost
{
public:
intgetCost(int area)
{
return area *70;
}
};

// Derived class
classRectangle:publicShape,publicPaintCost
{
public:
intgetArea()
{
return(width * height);
}
};

int main(void)
{
RectangleRect;
int area;

Rect.setWidth(5);
Rect.setHeight(7);

area=Rect.getArea();

// Print the area of the object.
cout<<"Total area: "<<Rect.getArea()<<endl;

// Print the total cost of painting
cout<<"Total paint cost: $"<<Rect.getCost(area)<<endl;

return0;
}
```

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

```
Total area:35
Total paint cost: $2450
```

5.

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

        Class M
{
                Protected:
                Int m;
                Public :
                Void get_M();
        };
        Class N
{
                Protected:
                Int n;
                Public:
                Void get_N();
        };
        Class p:public M, public N
        {
                Public:
                Void disply(void);
        };
        Void M ::get_m(int x)
        {
                m=x;
        }
```

```
Void N::get_n(int y)
{
        n=y;
}
Void P::disply(void)
{
Cout<<”m=”<<m<<endl;
Cout<<”n=”<<n<<endl;
Cout<<”m*n=”<<m*n<<endl;
}
int main()
{
        P p;
        p.get_m(10);
        p.get_n(20);
        p.display();
        return0;
}
```

OUTPUT

m=10
n=20
m*n=200

6.

```
#include<iostream.h>
#include<conio.h>
class B
{
int a;
public:
int b;
voidget_ab();
intget_a();
voidshow_a();
};
class D:private B
{
int c;
public:
voidmul();
void display();
};
void B::get_ab()
{
cout<<"Enter Values for a and b";
cin>>a>>b;
}
int B::get_a()
{
return a;
}
void B::show_a(){
                        cout<<"a= "<<a<<"\n”;
                }
void D::mul()
{
                        get_ab();
                        c=b*get_a();
        }
void D::display()
{
show_a();
cout<<"b= "<<b<<"\n”;
cout<<"c= "<<c<<"\n\n";
}
void main()
{
                        clrscr();
                        D d;
                        d.mul();
                        d.display();
                        d.mul();
                        d.display();
                        getch();
```

```
}
```

OUTPUT

A=5
A=5
B=10
C=50

A=5
B=20
C=100

7.

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

class student
{




        public:
        intrno;
        //float per;
        char name[20];
        voidgetdata()
        {
                cout<<"Enter RollNo :- \t";
                cin>>rno;
                cout<<"Enter Name :- \t";
                cin>>name;

        }

};
class marks : public student
{
public:
        int m1,m2,m3,tot;
        float per;
        voidgetmarks()
        {
                getdata();
                cout<<"Enter Marks 1 :- \t";
                cin>>m1;
                cout<<"Enter Marks 2 :- \t";
                cin>>m2;
                cout<<"Enter Marks 2 :- \t";
                cin>>m3;
        }
        void display()
        {
                getmarks();
                cout<<"Roll Not \t Name \t Marks1 \t marks2 \t Marks3 \t Total \t
Percentage";

        cout<<rno<<"\t"<<name<<"\t"<<m1<<"\t"<<m2<<"\t"<<m3<<"\t"<<tot<<"\t"<<per;
        }
};
void main()
{
        studentstd;
        clrscr();
        std.getmarks();
        std.display();
        getch();
}
```

8.

```
#include <iostream.h>
using namespace std;
class A {
int data;
public:
void f(intarg) { data = arg; }
int g() { return data; }
};

class B {
public:
A x;
};

int main() {
   B obj;
obj.x.f(20);
cout<<obj.x.g() <<endl;
//   cout<<obj.g() <<endl;
}
```

9.

```
#include<iostream.h>
usingnamespacestd;
classArea
{
public:
floatarea_calc(floatl,float b)
{
return l*b;
}
};

classPerimeter
{
public:
floatperi_calc(floatl,float b)
{
return2*(l+b);
}
};

/* Rectangle class is derived from classes Area and Perimeter. */
classRectangle:privateArea,privatePerimeter
{
private:
float length, breadth;
public:
Rectangle(): length(0.0), breadth(0.0){}
voidget_data()
{
cout<<"Enter length: ";
cin>>length;
cout<<"Enter breadth: ";
cin>>breadth;
}

floatarea_calc()
{
/* Calls area_calc() of class Area and returns it. */

returnArea::area_calc(length,breadth);
}

floatperi_calc()
{
/* Calls peri_calc() function of class Perimeter and returns it. */

returnPerimeter::peri_calc(length,breadth);
}
};

int main()
{
Rectangle r;
```

```
r.get_data();
cout<<"Area = "<<r.area_calc();
cout<<"\nPerimeter = "<<r.peri_calc();
return0;
}
```

10.

PROGRAM: PAYROLL SYSTEM USING SINGLE INHERITANCE

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

class emp
{
   public:
     int eno;
     char name[20],des[20];
     void get()
     {
               cout<<"Enter the employee number:";
               cin>>eno;
               cout<<"Enter the employee name:";
               cin>>name;
               cout<<"Enter the designation:";
               cin>>des;
     }
};

class salary:public emp
{
     float bp,hra,da,pf,np;
   public:
     void get1()
     {
               cout<<"Enter the basic pay:";
               cin>>bp;
               cout<<"Enter the Human Resource Allowance:";
               cin>>hra;
               cout<<"Enter the Dearness Allowance :";
               cin>>da;
               cout<<"Enter the Profitability Fund:";
               cin>>pf;
     }
     void calculate()
     {
               np=bp+hra+da-pf;
     }
     void display()
     {

cout<<eno<<"\t"<<name<<"\t"<<des<<"\t"<<bp<<"\t"<<hra<<"\t"<<da<<"\t"<<pf<<"\t"<<np<
<"\n";
     }
};

void main()
{
    int i,n;
    char ch;
    salary s[10];
    clrscr();
    cout<<"Enter the number of employee:";
    cin>>n;
    for(i=0;i<n;i++)
    {
               s[i].get();
               s[i].get1();
               s[i].calculate();
    }
    cout<<"\ne_no \t e_name\t des \t bp \t hra \t da \t pf \t np \n";
    for(i=0;i<n;i++)
    {
               s[i].display();
    }
    getch();
}
```

```
Output:
Enter the Number of employee:1
Enter the employee No: 150
Enter the employee Name: ram
Enter the designation: Manager
Enter the basic pay: 5000
Enter the HR allowance: 1000
Enter the Dearness allowance: 500
Enter the profitability Fund: 300

E.No   E.name   des      BP    HRA   DA   PF     NP
150    ram      Manager  5000  1000  500  300    62
```

Exercise

Show the output of the following programs:

Program 1:

```
// friend functions
#include <iostream.h>
usingnamespacestd;

class CRectangle {
int width, height;
public:
voidset_values (int, int);
int area () {return (width * height);}
friendCRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
width = a;
height = b;
}

CRectangle duplicate (CRectanglerectparam)
{
CRectanglerectres;
rectres.width = rectparam.width*2;
rectres.height = rectparam.height*2;
return (rectres);
}

int main () {
CRectanglerect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
cout<<rectb.area();
return 0;
}
```

Program 2:

```
// friend class
#include <iostream.h>
using namespacestd;

class CSquare;

class CRectangle {
int width, height;
public:
int area ()
      {return (width * height);}
void convert (CSquare a);
};

class CSquare {
private:
```

```
int side;
public:
voidset_side (int a)
      {side=a;}
friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
width = a.side;
height = a.side;
}

int main () {
CSquaresqr;
CRectanglerect;
sqr.set_side(4);
rect.convert(sqr);
cout<<rect.area();
return 0;
}
```

Program 3:

```
// derived classes
#include <iostream.h>
usingnamespacestd;

classCPolygon {
protected:
int width, height;
public:
voidset_values (int a, int b)
{ width=a; height=b;}
  };

classCRectangle: publicCPolygon {
public:
int area ()
{ return (width * height); }
  };

classCTriangle: publicCPolygon {
public:
int area ()
{ return (width * height / 2); }
  };

int main () {
CRectanglerect;
CTriangletrgl;
rect.set_values (4,5);
trgl.set_values (4,5);
cout<<rect.area() <<endl;
cout<<trgl.area() <<endl;
return 0;
}
```

Program 4:

```
// constructors and derived classes
#include <iostream.h>
usingnamespacestd;

class mother {
public:
mother ()
{ cout<< "mother: no parameters\n"; }
mother (int a)
{ cout<< "mother: int parameter\n"; }
};



class daughter : public mother {
public:
```

```
daughter (int a)
{ cout<< "daughter: int parameter\n\n"; }
};

class son : public mother {
public:
son (int a) : mother (a)
{ cout<< "son: int parameter\n\n"; }
};

int main () {
daughtercynthia (0);
sondaniel(0);

return 0;
}
```

Program 5:

```
// multiple inheritance
#include <iostream.h>
usingnamespacestd;

classCPolygon {
protected:
int width, height;
public:
voidset_values (int a, int b)
{ width=a; height=b;}
  };

classCOutput {
public:
void output (int i);
  };

voidCOutput::output (int i) {
cout<< i <<endl;
  }

classCRectangle: publicCPolygon, publicCOutput {
public:
int area ()
{ return (width * height); }
  };

classCTriangle: publicCPolygon, publicCOutput {
public:
int area ()
{ return (width * height / 2); }
  };
int main () {
CRectanglerect;
CTriangletrgl;
rect.set_values (4,5);
trgl.set_values (4,5);
rect.output (rect.area());
trgl.output (trgl.area());
return 0;
}
```

Lesson -- 5

Polymorphism

In this lesson we will learn about Polymorphism, virtual function, static and dynamic binding, abstract base, virtual destructor and virtual table.

5.1 Polymorphism

The meaning of polymorphism is "One Name, Multiple Forms". It allows us to create multiple functions with one name in a program (Function Over loading).

It also allows us to do operator Overloading which means showing different behaviour under different instances.

Polymorphism is of two types :

* Compile Time Polymorphism
* Run Time Polymorphism

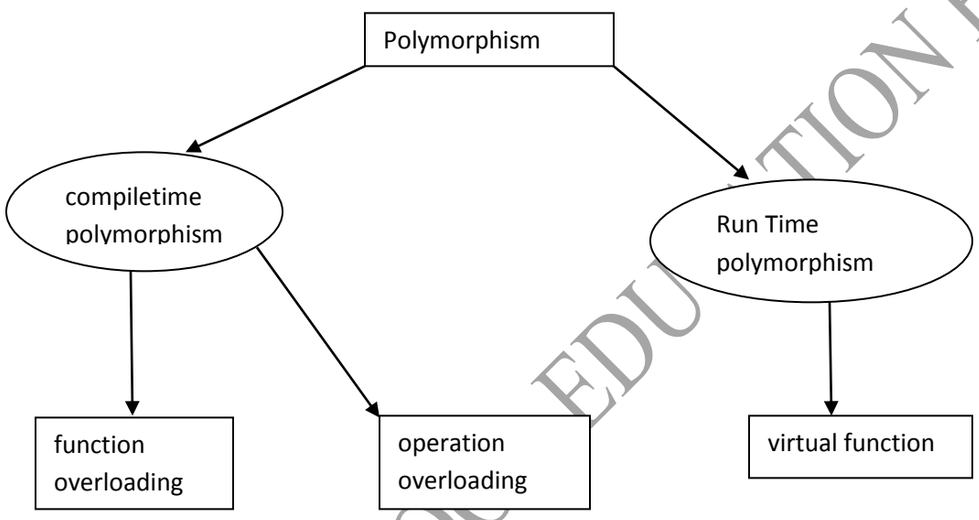


Diagram 5.1 Polymorphism

5.1.1 Static Binding

In this, any function (procedure) related code is known to compiler at the time of compilation.

5.1.2 Dynamic Binding

In this, any function related code is known to compiler at run time.

5.2 virtual function

As it has been proved earlier that polymorphism has one more characteristic that it can answer one message from related objects of different classes with different ways.

Therefore a single pointer variable is require, which can refer objects of different collectively.

Here we use base class pointer to refer derived objects. When we use both base and derived class with one functions name then virtual word is used as prefix to the functions declared in base class.

When a virtual function is created, then C++ tells instead of taking care of type of pointer use function during run time. Which is pointed by base pointer with object of function. We can run different virtual functions this way.

Program 5.2

VIRTUAL FUNCTIONS

```
# include <iostream.h>
class Base
{
```

```
    public :
    void display( ) {cout << ‘‘\n Display base’’ : }
    virtual void show ( ) {cout << ‘‘\n showbase’’ ;}
};
class Derived : public Base
{
    public :
    void display ( ) {cout << ‘‘\n Display derived’’ ;}
    void show ( ) {cout << ‘‘\n Show derived’’ ;}
};

    int main( )
{
    Base B;
    Derived D;
    Base * btpr ;
cout << ‘‘\n btpr Points to Base \n’’;
btpr = & B ;
btrp -> display ( ) ;   //Calls Base Version
btpr -> show ( ) ;     //Calls Base Version
cout << ‘‘\n \n btpr Points to Derived \n’’;
btpr = & 0;
btpr -> display ( );   //Calls Base Version
btpr -> show ( ) ;     //Calls Derived Version
return 0;
}
```

The output of Program would be

```
bptr Points to Base
Display base
Show base
btpr Points to Derived
Display base
Show derived
```

5.2.1 **Pure virtual Function**

A Pure virtual function is a function which is declared in base class but does not contain base class related definition. A virtual function which is equal to zero, is called a pure virtual function.

5.3 **Abstract Base classes**

A class in which there are pure virtual function, is called abstract base class. Here it is to remember, a class having pure virtual function, can not be used to declare any object.

5.3.1 **Interfaces in C++**

An Interface defines ability or behavior of class without knowing whether c++ class is executed or not executed.

c++ Interface are executed using abstract class. A class can be made abstract by declaring any function of a class as “pure virtual function”

specifying pure virtual function “= O” and its declaration of

Declaration

```
class Box
{
 public :
    // Pure virtual function
    virtual double get volume ( ) = 0;
    private :
    double length;   //length of a box
    double breadth;  //breadth of a box
    double height;   //height of a box
};
```

Example 5.3

Abstract class Example

Consider the following example where parent class provides an interface to the base implement a function called getArea ( ) :

```
# include <iostream.h>
using namespace std;
// Base class
class Shape
{
public :
    // pure virtual function providing interface framework,
    virtual int getArea ( ) = 0 ;
    void setWidth (int w)
    {
        Width = w;
    }
    void setHeight (int h)
    {
        height = h'
    }
protected :
    int width ;
    int height ;
};
//Derived classes
class Rectangle : public Shape
{
public :
    int getArea ( )
    {
        return (width * height) ;
    }
};
class Triangle : public Shape
{
public :
    int getArea (*)
    {
        return (width * height)/2;
    }
};
int main (void)
{
    Rectangle Rect;
    Triangle Tri;
    Rect.setWidth (3) ;
    Rect.setHeight (7) ;
    // Print the area of the object.
    cout << ''Total Rectangle area : '' << Rect.get Area ( ) << end1;
    Tri.setWidth (5) ;
    Tri.setHeight (7) ;
    // Print the area of the object.
    cout << ''Total Triangle area : ''Tri.getArea ( ) << endl;
    return 0;
}
```

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

Total Rectangle area : 35

Total Triangle area : 17

5.4 **virtual Table**

Virtual Table is also known as V table. Infact Virtual Table is very simple but it is bit tough to define it in the words.

first of all, each class which is using virtual function, is provided its own virtual table, This table is basically a static array which created by compilers during compile time. When virtual function is called by object of class, then virtual table takes its one entry. Each entry of this table is a function pointer, which points to derived function of a class.

Example 5.4

```
1
2  class Base
3  {
4  public :
5  virtual void : function1 ( ) { };
6  virtual void function2 ( ) { };
7  };
8  class D1 : public Base
9  {
10 public :
11 virtual void function ( ) { } ;
12 };
13 class D2 : public Base
14 {
15 public :
16 virtual void function2 ( ) { } ;
17 };
18
```

In this example there are 3 classes. therefor compiler will create 3 virtual tables. One is for base class, one for D1 and one for D2.

5.5 **virtual Destructor**

As we know, In c++ destructors are used to free memory space. We show destructor by using tilde (~) character. While defining virtual function, virtual (~) is written before tilde ~ character.

Need for virtual destructor in c++ can be understood through example given below

example 5.5.1 without virtual destructor

```
# include <iostream.h>
class Base
{
    public :
    Base ( ) {cout << ‘‘Constructing Base’’;}
    //this is a destructor;
    ~ Base ( ) {cout << ‘‘Destroying Base’’; }
};
class De
{
    public :
    Derive ( ) {cout << ‘‘Constructing Derive’’;}
    ~ Derive ( ) {cout <<‘‘Destroying Derive’’;}
};
void main ( )
{
    Bass * baseptr = new Derive ( ) ;
    delete base Ptr ;
}
```

ਅਸੀਂ ਇਸ ਬਾਰੇ ਵਿਸਥਾਰ ਨਾਲ ਅੱਗੇ ਪੜ੍ਹਾਗੇ।

Output :

Constructing Base

Constructing Derive

Destroying Base

We can see through this example that constructors are called in right order. When object pointer of derive class are created in main function.

But there comes a major problem with code when we delete ‘ base ptr ’, which is destructor for “ Derive class, is not called.

example 5.5.2 with virtual destructor

```
class Base
{
   public :
   Base ( ) {cout << ‘‘Constructing Base’’;}
   //this is a virtual destructor ;
   virtual ~ Base ( ) {cout << ‘‘Destroying Base’’ ;}
};
```

Note :- Derived class destructor is called before base class.

Therefore it can be clearly seen that why a virtual destruction is required and also how do they work ?

5 .6 **Vector**

vector are those continuous container which specifies arrays and changes its size

vector uses dynamically allocated array to store its elements. It is required to reallocate this array because when new elements are added, Its size increases and a new array is allocated so that all elements can be moved to it.

It is very time consuming process. Therefore instead of using these, extra storage is allocated to vector containers so that added elements can be stored there. If it is compared to array, Vectors consume more memory.

Example 5 .6

```
// constructing vectors
# include <iostream.h>
# include <vector>

int main ( )
{
unsigned int i ;
// constructors used in the same user as described above :
std :: vector < int > first ;    //empty vector of  ints
std :: vector < int > second (4,100) ;   //four ints with value 100
std :: vector < int > third (second.begin ( ), second.end( ) );  // iterating through second
std :: vector < int > fourth (third) ;   // a copy of third   the iterator constructor can also be
                                         // used to construct from arrays :
int myints [ ] = {16, 2, 77, 29};
std :: vector < int > fifth (myints, myints + size of (myints) / sizeof (int) );

std :: cout << ‘‘The contents of fifth are :’’;
for (std :: vector < int > :: iterator :: fifth.begain) ; it ! - fifth.end ( );++it)
std :: cont << ‘ ’ << * it ;
std :: cout << ‘\n’ ;
return 0 ;
 }
```

Output :

The contents of fifth are : 16 2 77 2

Points to be remembered :

1. Polymorphism means one name, multiple forms.
2. There are two types of Polymorphism.
3. An interface defines ability and behaviour of c++ class.
4. Virtual table is also known as “Vtable”
5. Destructors are used to free memory space.

Exercise

1. *Fill in the blank*

1. Polymorphism means one name ----------
2. Each class, using virtual function, have its----------
3. ----------is specified by (~) character/Mark.
4. A function which is defined in base class but having no relation with base class, called----------
5. ----------are contiguous containers.

2. Short answer questions

1. Write a note on Polymorphism.
2. What are abstract base classes.
3. What is Pure and virtual functions.
4. What is the difference between static binding and Dynamic binding.
5. What are vectors.
6. Explain virtual functions with example.

Lab Activity

Program 1:

```
// pointers to base class
#include <iostream.h>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };

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 * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Program 2:

```
// virtual members
#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 * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Program 3:

```
// abstract class CPolygon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Program 4:

```
// abstract base class
#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 (void) =0;
  };
```

```
class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

Program 5:

```
// pure virtual members can be called
// from the abstract base class
#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 (void) =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

Program 6:

```
// dynamic allocation and polymorphism
#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 (void) =0;
```

```
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };
class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };
int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Lesson -- 6

Input/Output with Files

6.1 C++ provides class is given below which perform input and output on and from files.

- ofstream : To write stream class on files
- ifstream : To read stream class from files.
- ifstream : To read/write stream classes on/ from files.

These classes are taken from istream and ostream classes directly or indirectly. We have already used objects whose types are these classes.

cin : is an object of a class istream. and
cout : is an object of a class ostream
Therefore We are using those classes which are related to file stream
let us see example 6.1

```
1  // basic file operations
2  # include <iostream.h>
3  # include <fstream.h>
4  using namespace std;
5
6  int main ( ) {        [file example.txt] writing this to a file
7  of stream my file;
8  myfile.open (‘‘example.txt’’);
9  myfile <<‘‘writing this to a file. \n’’;
10 myfile . close ( ) ;
11 return 0;
12 }
```

one code makes one file. It is called example. txt which is inserted once, just the way we do with cout. But in this used is stream my file. Let us follow it step by step.

6.2 **Open a File**

first operation is done on an object of one of these classes, which is combined to read file. This process is known as open a file.

A file which is of stream object, we use member function to open it.

```
Open (file name, mode) ;
```

where file name is null-terminated character which is type const char and specifies file name. Which is to be open.

ios :: in open for input operations
ios :: out open for output operations
ios :: binary open in binary mode
ios :: ate set the initial position at the end of file
ios : app the current content of the file.
ios :: trunc if the file opened for output operations already existed before

Modes of flags are optional parameter is given below.

These all flags can be combined using bitwise operator OR() example :-

Example:--

```
1 of stream myfile ;
2 my file . open (‘‘example.bin’’, ios :: out / ios :: app ios :: binary) ;
```

6.3 **Closing a file**

When we complete our input/output operations on file. Then we can close them.To do this stream member function close () is called so that this resources would be available again.

```
my file.close ( ) ;
```

once a member function is called then the stream object are used to open another file and to open other process, file gets available again.

6.4 Text Files

Text files are those where we do not use ios ::binary flag in open mode.

These files are designed to store text and all the transformation formatting done values input or output are affected Data output operations are used in this some way we use with cout:

```
1. // 6.4 Writing on a text file
2. # include <iostream.h>          [file example.txt]
3. # include <fstream.h>          This is a line. This is another line.
4. using namespace std;
5.
6. int main ( ) {
7. ofstream my file (‘‘example txt’’) ;
8. if (my fle.is-open ( ) )
9. {
10. my file << ‘‘This is a line \n’’ ;
11. my file << ‘‘This is another line \n’’;
12. my file close ( ) ;
13. }
14. else cout << ‘‘unable to open file’’ ;
15. return 0 ;
16. }
```

Data input runs from file the same way we do with cin.

```
1. // reading a text file
2. # include <iostream.h>
3. # include <fstream.h>
4. # include <string.h>
5. using namespace std;
6.
7. int main ( ) {
8. string line ;
9. ifstream myfile (‘‘example.txt’’);
10. if (my file is – open ( ) )
11. {
12. While (my file good ( ) )  This is line.
13. {                This is another line.
14. getline (my file, line) ;
15. cout << line << endl ;
16. }
17. my file, close ( ) ;
18. }
19.
20. else cout << ‘‘unable to open file’’ ;
21.
22. return 0 ;
23. }
```

This last example reads a text file and print its contents on screen. It is to be noted that how do we used a new member function good (), which is true in resultant case of stream Input/Output Operation.

6.5 Checking State Flags

A member function good () which checks whether stream is prepared for input/output Operations. One more member function is present that checks that all return states of stream value

bad ( )

It returns true if all Reading and Writing operation got failed.

fail ( )

It returns true just like bad (), but when format error occurs, an alphabetical character is extracted by reading integer number.

eof ( )

It returns true if file is opened to read and has reached at the end

good ( )

This is best generic state flag which returns false, for those causes it has returned true earlier.

to reset the state flags. which have been checked by member function We can use clear () member function. Which does not contain any parameter.

6.6 **get and Put stream Pointers**

All i/o stream consists of atleast one internal stream Pointer. If stream has one pointer, get pointer "like istream which points towards the elements.of next input operation to be read.

In the last, fstream that originated from both get and put pointer which is again came from iostream.this internal stream i.e pointer stream points towards the reading and Righting locations can be manipulated through the member functions given below : -

6.6.1 **tellg ( ) and tellp ( )**

There is no parameter of these two member functions and return value of member type of POS-Type. Which is of integer data type, At current position it is considered as get stream pointer in case of integer data type tellg () and in other case put stream pointer of tell p ().

6.6.2 **seekg( ) and seekp( )**

These functions allowus to change the position of get and put stream pointer. Both functions, are overloaded by different prototypes.

* First Prototype is
  seek g (Position) ;
  seek P (Position) ;
* Second Prototype is
  seek g (offset, direction) ;
  seek P (offset, direction) ;

following example 6.6.2member function which is to get size of file.

```
// obtaining file size
# include <iostream.h>
# include <fstream.h>
using namespace std;
int main ( )
{
long begin end ;
if stream my file (‘‘example.txt’’); size is : 40 bytes
begin = mgfile.tellg ( );
my file.seekg (0, ios :: end) ;
end = my file.tell g ( ) ;
myfile.close ( ) ;
cout << ‘‘size is :’’ << (end-begin) << ‘‘bytes \n’’ ;
return 0;
}
```

6.7 **Stream Manipulators)**

Stream Manipulators are used for formatting. For example

- To set field width
- Precision
- To unset format flags
- To insert new line in output stream and to flush stream.
- To insert NULL character in output stream.
- To skip white space

6.8 Stream Base :-

6.8.1 hex. To set base to hexadecimal base 16.
6.8.2 oct. To set base to octal base 8.
6.8.3 Dec. To reset stream to decimal.

6.8.4 Set. base()This is a parameterized manipulator which takes 10,8 or 16 as parameter so that integer be printed on that base. For example set base (16) will work as hex operator.

6.8.5 **flesh :-**It is used to flush output buffer before beginning the process.

6.8.6 **endl.** It prints a newline and flushes output buffer.

6.8.7 **width( ).** This field is used to set width. If the entered value is smaller than field width, then It inserts fill characters as padding.

6.8.8 **fill( ).** This is used to fill, fill character when we were using width function.

6.8.9 **setw( ).** This is a parameterized manipulator and works like width () function.

6.8.10 set file(): it is a prameterized manuplator it works like file() function

6.9 **Binary Files**

It is not correct to do Data extraction, Insertion and function operator (with getline input or output) with in Binary files. Therefore we do not need to change any data. for example There is no need to give space, newline code to data separately Two member functions are designed in file stream. Which takes input and output Binary data respectively

* Write and Read.

First (write) is the member function of ostream , which is taken from ofstream and (read) istream is member function of istream which is taken from ifstream

the object of class ifstream contains both members both the members and their prototype are as follows :-

write (memory_block, size);

read (memory_block, size);

example : 6.9

```
// reading a complete binary file
# include <iostream.h>
# include <fstream.h>
using namespace std;
ifstream :: POS-type size ;
char * memblock ;
int main ( )
 {
      ifstream file (‘‘example.bin’’,
      ios :: in/ios :: binary / ios :: ate) ;
       if (file . is - open ( ) )
{
size = file.tell g ( ) ;     the complete file content is in memory
memblock = new char [size] ;
file.seekg (0, ios :: beg) ;
file.read (memblock, size) ;
file.close ( );

cout << ‘‘the complete file content is in
memory’’;
delete [ ] memblock ;
}
else cout << ‘‘unable to open file’’;
return 0;
}
```

In this example whole file is read and store in memory block.

6.10 **Buffers and Synchronization**

When we operate on file stream then they are combined with internal buffer which is of type **streambuf** these buffers are block of memory which are intermediary between. **Stream** and **physical file**.

for example when ofstring related member function **put** is called, words directly related to string are not written on physical file, besides its words enter in intermediate buffer of stream.

When buffer is flushed, if data stored in it is of output stream, it is written in physical medium. If this data is of input stream, or it gets totally free. this process is called synchronization and this occurs under situations give below

1. When file is closed.
2. When buffer gets full.
3. When manipulators are used explicitly on some manipulator stream, external synchronization occurs.These manipulators are flush and end.

4.When member function sync () is called explicitly with member functions synchronization occurs during that time.

Points to be Remember:

1. To write ofstream class is on file, to read from file, and fstream class, to read and write from class are used .

Member function, open () and close () functions are used to open and close files.

3. all Istream objects consist of atleast one Internal stream Pointer.
4. Stream manipulators are used for formatting.
5. Write and Read are those two member functions which takes input and output binary data respectively.
6. Buffer is a block of memory which are intermediate between stream and physical.

Exercise

1. ***fill in the blanks***

1. ---------- stream class is used to write on files.
2. ---------- member function is used to open a file
3. Stream manipulator prints are used for ----------
4. ---------- manipulator prints a newline.
5. ---------- and ---------- functions allows us to change the position of get and put stream pointer

2.True and false

Set W () and Width () function are used to set width.
2. hex sets base to 8.
3. If all reading and Writing operations get failed then bad () returns false value.
4. When buffer gets full, synchronization process occurs.

3. Short answered questions.

1. Write difference between bad ()and good () ?
2. Write a brief note on text files ?
3. What are stream manipulators ?
4. What do you mean by buffer ?
5. Write about hex, oct and Dec manipulators ?

Lab Activity

Example 1

```
* Program to create a file and write some data on the file */
#include <stdio.h>
#include <stdio.h>
main( )
{
     FILE *fp;
char stuff[25];
int index;
fp = fopen("TENLINES.TXT","w"); /* open for writing */
strcpy(stuff,"This is an example line.");
for (index = 1; index <= 10; index++)
       fprintf(fp,"%s Line number %d\n", stuff, index);
fclose(fp); /* close the file before ending program */
}
```

Example 2

```
/* Program to display the contents of a file on screen */
#include <stdio.h>
void main()
{
   FILE *fopen(), *fp;
int c;
fp = fopen("prog.c","r");
   c = getc(fp) ;
while (c!= EOF)
   {
              putchar(c);
              c = getc(fp);
   }
```

Example 3

```
#include <stdio.h>
int main()
{
 FILE *fp;
file = fopen("file.txt","w");
 /*Create a file and add text*/
fprintf(fp,"%s","This is just an example :)"); /*writes data to the file*/
fclose(fp); /*done!*/
return 0;
}
```

Example 4

```
#include <stdio.h>
int main()
{
    FILE *fp
file = fopen("file.txt","a");
fprintf(fp,"%s","This is just an example :)"); /*append some text*/
fclose(fp);
return 0;
}
```

Example 5

```
#include <stdio.h>
main( )
   {
     FILE *fp;
char c;
funny = fopen("TENLINES.TXT", "r");
if (fp == NULL)
              printf("File doesn't exist\n");
else {
do {
       c = getc(fp); /* get one character from the file
       */
putchar(c); /* display it on the monitor
```

Example related to iostream

Program 1

```
#include <iostream.h>

using namespace std;

int main( )
{
charstr[] = "Hello C++";

cout<< "Value of str is : " <<str<<endl;
}
```

Program 2

```
#include <iostream.h>
 int main()
{
    usingnamespacestd;
    // First we'll use the insertion operator on cout to print text to the
monitor
    cout<< "Enter your age: "<<endl;

    // Then we'll use the extraction operator on cin to get input from the
user
    intnAge;
    cin>>nAge;

    if(nAge<= 0)
    {
```

```
        // In this case we'll use the insertion operatior on cerr to print an
error message
        cerr<< "Oops, you entered an invalid age!"<<endl;
        exit(1);
    }

    // Otherwise we'll use insertion again on cout to print a result
    cout<< "You entered "<<nAge<< " years old"<<endl;
return0;

}
```

Streamoutput programexample:

Program 3

```
//string output using <<
#include <stdlib.h>
#include <iostream.h>
void main(void)
{
cout<<"Welcome to C++ I/O module!!!"<<endl;
cout<<"Welcome to ";
cout<<"C++ module 18"<<endl;
//endl is end line stream manipulator
//issue a new line character and flushes the output buffer
//output buffer may be flushed by
cout<<flush;
commandsystem("pause");
}
```

Program 4

```
//concatenating <<
#include <stdlib.h>
//for system(), if compiled in some compiler
//such as Visual Studio, no need this stdlib.h
#include <iostream.h>
void main(void)
{
int p = 3, q = 10;
cout<< "Concatenating using << operator.\n"
<<"-------------------------------"<<endl;
cout<< "70 minus 20 is "<<(70 - 20)<<endl;
cout<< "55 plus 4 is "<<(55 + 4)<<endl;
cout<<p<<" + "<<q<<" = "<<(p+q)<<endl;
system("pause");
}
```

Stream input program example:

Program 5

```
#include <stdlib.h>
#include <iostream.h>
void main(void)
{
int p, q, r;
cout<< "Enter 3 integers separated by space: \n";
cin>>p>>q>>r;
//the >> operator skips whitespace characters such as tabs,
//blank space and newlines. When eof is encountered, zero (false)
//is returned.
cout<<"Sum of the "<<p<<","<<q<<" and "<<r<<" is = "<<(p+q+r)<<endl;
system("pause");
}
```

Program 6

```
//using hex, oct, dec and setbase stream manipulator
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
int p;
cout<<"Enter a decimal number:"<<endl;
cin>>p;




cout<<p<< " in hexadecimal is: "
<<hex<<p<<'\n'
<<dec<<p<<" in octal is: "
<<oct<<p<<'\n'
<<setbase(10) <<p<<" in decimal is: "
<<p<<endl;
cout<<endl;
system("pause");
}
```

Floating-point Precision

Program 7

```
//using precision and setprecision
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
void main(void)
{
doubletheroot = sqrt(11.55);
cout<<"Square root of 11.55 with various"<<endl;
cout<<" precisions"<<endl;
cout<<"--------------------------------"<<endl;
cout<<"Using 'precision':"<<endl;
for(intpoinplace=0; poinplace<=8; poinplace++)
{
cout.precision(poinplace);
cout<<theroot<<endl;
}
cout<<"\nUsing 'setprecision':"<<endl;
for(intpoinplace=0; poinplace<=8; poinplace++)
cout<<setprecision(poinplace)<<theroot<<endl;
system("pause");
}
```

Field Width

Program 8

```
//using width member function
#include <iostream.h>
#include <stdlib.h>
void main(void)
{
int p = 6;
char string[20];
cout<<"Using field width with setw() or width()"<<endl;
cout<<"---------------------------------------"<<endl;
cout<<"Enter a line of text:"<<endl;
cin.width(7);
while (cin>>string)
{
cout.width(p++);
cout<<string<<endl;
```

```
cin.width(7);
//use ctrl-z followed by return key or ctrl-d to exit
}
system("pause");
}
```

Trailing Zeroes and Decimal Points

Program 9

```
///Using showpoint




//controlling the trailing zeroes and floating points
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
void main(void)
{
cout<<"Before using the ios::showpoint flag\n"
<<"-----------------------------------"<<endl;
cout<<"cout prints 88.88000 as: "<<88.88000
<<"\ncout prints 88.80000 as: "<<88.80000
<<"\ncout prints 88.00000 as: "<<88.00000
<<"\n\nAfter using the ios::showpoint flag\n"
<<"-----------------------------------"<<endl;
cout.setf(ios::showpoint);
cout<<"cout prints 88.88000 as: "<<88.88000
<<"\ncout prints 88.80000 as: "<<88.80000
<<"\ncout prints 88.00000 as: "<<88.00000<<endl;
system("pause");
}
```

Manipulators

Program 10

```
//using setw(), setiosflags(), resetiosflags() manipulators
//and setf and unsetf member functions
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
void main(void)
{
long p = 123456789L;
//L - literal data type qualifier for long...
//F - float, UL unsigned integer...
cout<<"The default for 10 fields is right justified:\n"
<<setw(10)<<p
<<"\n\nUsing member function\n"
<<"--------------------\n"
<<"\nUsingsetf() to set ios::left:\n"<<setw(10);
cout.setf(ios::left,ios::adjustfield);
cout<<p<<"\nUsingunsetf() to restore the default:\n";
cout.unsetf(ios::left);
cout<<setw(10)<<p
<<"\n\nUsing parameterized stream manipulators\n"
<<"---------------------------------------\n"
<<"\nUsesetiosflags() to set the ios::left:\n"
<<setw(10)<<setiosflags(ios::left)<<p
<<"\nUsingresetiosflags() to restore the default:\n"
<<setw(10)<<resetiosflags(ios::left)
<<p<<endl;
system("pause");
}
```

Program 11

```
/using setw(), setiosflags(), showpos and internal
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
void main(void)
{
cout<<setiosflags(ios::internal | ios::showpos)
<<setw(12)<<12345<<endl;
system("pause");
```

Exercise

Point out the errors if any

```
   1. // writing on a text file
#include <iostream.h>
#include <fstream.h>
usingnamespacestd;

int main () {
ofstreammyfile ("example.txt");
if (myfile.is_open())
  {
myfile<< "This is a line.\n";
myfile<< "This is another line.\n";

  }
elsecout<< "Unable to open file";
return 0;
}


2.// obtaining file size
#include <iostream.h>
#include <fstream.h>
usingnamespacestd;

int main () {
longbegin,end;
ifstreammyfile ("example.txt");
begin = myfile.tellg();
myfile.seekg (0, ios::end);
end = myfile.tellg();
myfile.close();
cout<< "size is: " << (end-begin) << " bytes.\n";
return 0;
}
3./ basic file operations
#include <iostream.h>
#include <fstream.h>
usingnamespacestd;

int main () {
ofstreammyfile;
myfile.open ("example.txt");
myfile<< "Writing this to a file.\n";
myfile.close();

}
```

Lesson -7

Templates and Exception Handling

n this chapter we will learn about the following topics.

- Templates
- Class Templates
- Function Templates
- Overloading of Template function
- Exception Handling
- Name space, cast operator

7.1 Templates

This is a topic by which we can define generic classes and generic functions.

General programming is a process in which generic Data Types are used parameters Templates are used to create class or function family. Templates are also called Macro.

example 7.1 Program (Template)

```
template < class T >
class vector
{
   T * V ;      // Type T vector
   int size ;
public :
      vector (int m)
   {

      V = new T [size = m] ;
      fon (int i = 0; i<size, i++)
      V[i] = 0;
   }
   vector (T * a)
   {
      for (int i = 0, i < size ; i ++)
          v[i] = a[i];
   }
T operator * (vector & y)
{
      T sum = 0;
   for (int i = 0 ; i < size ; i++)
   sum + = this -> v[i] + y – v[i] ;
   return sum ;
   }
};
```

7.1.1 class Templates

As specified earlier, templates allow us to create generic classes. Method used to create this is given below :-

```
template < class T >
class classname
{
   //.....


   //class member specification
   //with anonymous type T
   //wherever appropriate
   //.....
};
```

A class which is created by using template is called template class.

7.1.2 Function Templates

We can define function templates like class templates. Which make function family (group) with different argument types.

format

```
template < class T >
returntype functioname (arguments of type T)
{
  //.....
  //Body of function
  //With type T
  //Wherever appropriate
  //.....
}
```

7.2 Overloading of Template Function

A Template function can be overloaded either by template function or by simple function. In this situation, overloading Resolution is achieved as given below

1. A simple function is should be called which has exact match.
2. A template function is should be called which is created from exact match.
3. Make use of Normal Overloading Resolution.
   Error message will be displayed if no match occur.

Example :

```
# include <iostream.h>
# include <string.h>
using namespace std,
template <class T>
void display (T  x)
{
    cout << ‘‘Template display :’’ << x << \n’’;
}
void display (int x)     //overloads the generic display( )
{
    cout << ‘‘Explicit display :’’ << x << ‘‘\n’’ ;
}
    int main ( )
    {
        display (100) ;
        display (12.34) ;
        display (‘C’) ;
        return 0;
    }
```

The output of Program is

```
Explicit display : 100
Template display : 12.34
Template display : C
```

7.3 Exception Handling

What are exceptions :-

Exceptions are those problems which occurs when program is run or executed.

Basics of Exception handling’s exceptions are of two types.

* Synchronous Exception
* Asynchronous exception

- Asynchronous exception

Synchronous exceptions include “out of-range index” and over flow arrays.

Asynchronous. exceptions are the arrays which happen from those events which are beyond the program control.

7.4 Exception Class

Exception Handling Mechanism in C++ is made of three keywords-try, throw, catch.

Try :- Try finds the errors or exception in block of statements.

Throw :- When a exception is found, To get rid of it throw statements is sent.

Catch :-Statements sent by throw statements, comes under catch statements. Catch tries to overcome this.

fig.

Try block throwing exception.....

```
.....
try
{
   .....
   throw exception;      // Block of statements which
   .....                 // detects and throws on exception
   .....
}
catch (type arg)         // catches exception
{
   .....
.....                    // Block of statements that
   .....                 // handles the exception
}
.....
.....
```

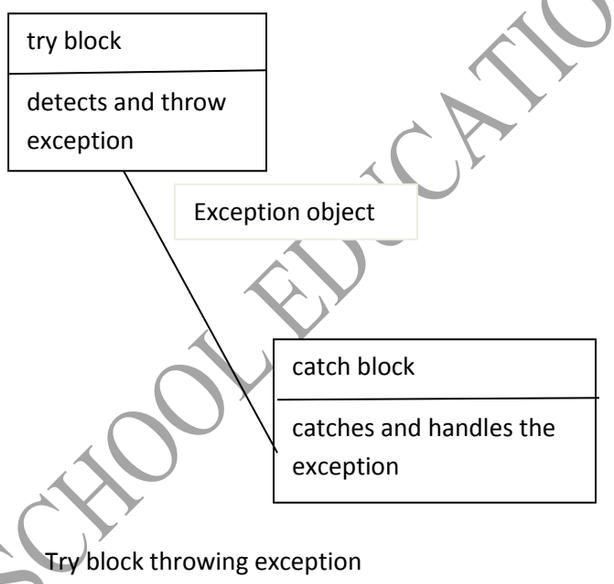


Fig. Try block throwing exception

7.5 **Name Space Concept**

In c++, We can define variables at different places for example. class, function, block etc. In C++, new keyword has been introduced-Name space which defines a variable globally.

c++ standard library is the best example of Name Space.

Using namespace std;

Namespace can be defined like any class. We can create our namespace.

The format of Namespace is given below.

```
namespace namespace_name
{
    // Declaration of
    // Variables, functions, class etc.
}
```

example 7.5

```
# include < iostream>
using namespace std ;
// Defining a namespace
```

```
namespace Name 1
{
    double x = 4.56 ;
    int m = 100 ;
    name space  Name 2  // Nesting namespace
{
    double  = 1.23 ;
    }
}
name space


{
    int m = 200 ;
}
int main ( )
{
    cout << ‘‘x = ’’ << Name1 :: x << ‘‘\n’’ ; // x is qualified
    cout << ‘‘m = ’’ << Name1 :: m << ‘‘\n’’ ;
    cout << ‘‘y =’’ << Name1 :: Name2 :: y << ‘‘\n’’ ;
                                                    // T is fully qualified.
```

7.6 **Cast operator**

Cast operator is used to convert a value of data type to another data type.It is necessary in the case where automatic conversion is not possible.

double x=double(m) //c++type casting

7.6.1 **The Static-Cast operator**

Static cast operator is also used for data type just like cast operator. It can be used to convert base class pointer to derived class pointer.

static_cast (type) (object)

example

```
int m = 10 ;
double x = static_cast < double > (m) ;
```

7.6.2 **The const_Cost operator**

Constant cast operator is used to override the data type of variable explicitly. Which means operator converts constant attribute of data type.

const_cost <type> (object)

7.6.3 **The reinterpret-cast operator**

If a pointer type object is to be converted into integer type, then reinterpret cast operator is used.

reinterpret_cast <type> (object)

7.6.4 **The dynamic-cast operator**

It is used to convert one data type to another during run time.

dynamic_cast <type> (object)

7.7 **Exception handling vs. Traditional Error Handling**

There was no built in facility to solve run time errors. Traditional Error handing method was used, in which following methodologies used to handle error is given below.

program

1. To prepare logic of program
2. To Write Pseudo code

for e.g.

2(1) Perform a Task
2(2) If the of proceeding works or task does not run correctly, apply process to find error.
2(3) Continue with next if proceeding works or dose task not proceeding well, proceed
3. It creates difficulty in reading, Editing and maintenance of program.
4. It affects, Impact Performance system.

This method (Traditional Error handling) can neither object oriented environment work to create application on large scale nor support oop enviornment.

Points to Remember:

1. Generic programming is a mechanism in which generic data type are used as parameter.
2. Templates are used to create class or function family.
3. Exceptions are those difficulties which occur during run time.
4. Name space defines a variable globally.
5. Cast operator is used to convert the value of one data type to another data type.

Exercise

1. **Fill in the blanks**

1. Templates allows to create ---------- classes.
2. Exceptions are of ---------- Types,
3. Exception handling in c++ is made up of ---------- and ---------- Keywords.

4.---------- operator can be used to convert base class pointer to derived class pointer.

2. **True of False**

1. We can not use variable in a program which is defined in namespace.
2. Template are used to make group of class and function.
3. The best example of Namespace is c++ standard Library.
4. A Template function can not be overloaded.

3. **Short answer questions**

1. What is generic programming ?
2. What is Exception? Write its types.
3. Define function Template ?
4. Write brief note on Cast Operator.
5. Write a note on Namespace.

4. **Long Answer Questions**

1. Explain the process of Namespace.
2. Explain process of exception handling with example?
3. How does Template created. Explain.

Lab Activity

Templates

Program 1:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Program 2:

```
// function template
#include <iostream.h>
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;
}
```

Program 3:

```
// function template II
#include <iostream.h>
using namespace std;

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

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

Program 4:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

Program 5:

```
// class templates
#include <iostream.h>
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;
}
```

Program 6

```
// template specialization
#include <iostream.h>
using namespace std;

// class template:
template <class T>
class mycontainer {

    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

Program 7:

```
// sequence template
#include <iostream.h>
using namespace std;

template <class T, int N>
```

```
class mysequence {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  mysequence <int,5> myints;
  mysequence <double,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

Exception Handling

Program 8:

```
// exceptions

#include <iostream.h>
using namespace std;

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << endl;
  }
  return 0;
}
```

Program 9:

```
try {
  try {
    // code here
  }
  catch (int n) {
    throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Program 10:

```
// standard exceptions
#include <iostream.h>
#include <exception>
using namespace std;
```

```
class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

Program 11:

```
// bad_alloc standard exception
#include <iostream.h>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

Namespaces

Program 12:

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

Program 13:

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

namespace first
```

```
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

Program 14:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

Program 15:

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```